MS 3.2 – Addressing Scalability: Scalability of open, close, flush CASE STUDY: CGNS Hotspot analysis of CGNS *cgp_open*

The HDF Group

This report addresses scalability issues (MS 3.2) associated with opening an HDF5 file which has a large quantity of metadata related to an application's file format specifications. More specifically, the metadata is stored in small HDF5 datasets, and all the processes read the content of the entire dataset.

As a case study, this report first details a performance issue with the API *cgp_open* in the CGNS library. Two solutions are then proposed to improve the performance: (1) through an HDF5 library modification (the *proc0-read-and-bcast* solution), and (2) a CGNS library modification (the use of compact dataset storage).

1 Rationale for the Investigation

1.1 Description of the Benchmark

The CGNS benchmark creates 128 bases with 32 zones in each base. The reported timing data is during **cgp_open** of an already created CGNS file. The majority of the **cgp_open** time is spent in **cgi_read**, and the remaining sections of the report break down the cascading calls in **cgi_read**. The number and size of the reads during **cgp_open** for this case is as follows:

SIZE OF ARRAY	TOTAL NUMBER OF READS	SIZE PER READ (BYTES)	TOTAL READ SIZE (BYTES)
2X1	128	8	1,024
3X3	4096	72	294,912
10X1	4096	10	40,960
1X1	1	4	4

Total read size is ~329K. The systems used in this report were: **Sky Bridge** (Cray, Sandia National Laboratories) and **Edison** (Cray, NERSC).

1.2 Benchmark Findings

In **cgi_read,** at line:

```
for (b=0; b<cg->nbases; b++) if (cgi_read_base(&cg->base[b])) return CG_ERROR;
```

Sky Bridge timing:

1 process	**24.4 s**
2048 processes	781 s

The average time spent in the above cgi_read_base call at:

cgi_read_base:

```
for (n=0; n<base->nzones; n++) {
          base->zone[n].id = id[n];
          base->zone[n].link = cgi_read_link(id[n]);
          base->zone[n].in_link = 0;
          if (cgi_read_zone(&base->zone[n])) return CG_ERROR;
      }
```

Sky Bridge timing:

1 process	**0.14 s**
2048 processes	5.95 s

The average time spent in cgi_read_zone at:

cgi_read_zone

```
   if (cgio_read_all_data(cg->cgio, node_id, data[0])) {
       cg_io_error("cgio_read_all_data");
       return CG_ERROR;
   }
```

Sky Bridge timing:

1 process	**0.012s**
2048 processes	5.46s

cgi_read_all_data calls the HDF5 APIs via ADFH_Read_All_Data.

The average time spent in ADFH_Read_All_Data in:

ADFH_Read_All_Data

```
if (H5Dread(did, mid, H5S_ALL, H5S_ALL, xfer_prp, data) < 0)

  set_error(ADFH_ERR_DREAD, err);

else

  set_error(NO_ERROR, err);
```

Sky Bridge timing:

1 process	**0.00361 s**
2048 processes	5.57 s

1.3 Summary of CGNS Benchmark

In general, this is a typical HDF5 use case where users augment the raw data with file format metadata as either attributes or small datasets. In the case of CGNS, it is the *base, zone* and *zone-type* metadata. While this means of organizing the data does not cause an issue when reading in serial (as the above results show), but for the parallel case, having all the processes read all of the same data is not scalable. One possible solution is to read and aggregate the metadata to fewer nodes and then do a broadcast of the data to the remaining processes or to open the file in serial, read the metadata with one process, and then have that process broadcast the metadata to all the other processes. The remaining sections of the report investigate various solutions.

1.4 MPI IO only benchmark Investigation

A benchmark was written to simulate the poor read scaling performance using only MPI IO. The benchmark writes an integer array of dimension 524,288 (2MB) and the file is closed. The file is opened, and *MPI_File_read_all* is called 8192 times by all the process reading the same data. A single read consists of a total of 64 bytes, and each read was offset from the previous read by 256 bytes.

The timing shows similar behavior as to that shown with CGNS on Sky Bridge:

1 process	**0.15 s**
2048 processes	470 s

Alternative options for reading the metadata:

(1) Read all the metadata on one process, then Bcast the metadata to the all the other processes.

(2) Read one metadata entry on one process, then Bcast that metadata to all the other processes. Repeat until all the metadata has been read (i.e., 8192 times for this case).

Option 0 (All processes read the metadata)	
1 process	0.15 s
2048 processes	474 s

Option 1 (One MPI_Bcast of the metadata)	
1 process	0.15 s
2048 processes	0.18 s

Option 2 (One MPI_Bcast per metadata entry)	
1 process	0.17 s
2048 processes	0.27 s

1.5 CGNS benchmark on GPFS

The benchmark in the main section of the program was run on cetus (mira) at ANL:

The average time spent in ADFH_Read_All_Data in:

ADFH_Read_All_Data

```
  if (H5Dread(did, mid, H5S_ALL, H5S_ALL, xfer_prp, data) < 0)

    set_error(ADFH_ERR_DREAD, err);

  else

    set_error(NO_ERROR, err);
```

cetus (ANL) timing:

1 process	**0.56 s**
2048 processes**	25.1 s

** job was killed before all the H5Dreads had completed.

2 Alternative Prototype Metadata Read Options

Two alternative read options were investigated:

- **read-proc0-and-bcast** – A new transfer property flag was added to the HDF5 library to enable reading the data on one process and then broadcasting the data to the other processes. CGNS was changed to use this option when reading data that is less than 2MB.
- **Compact Storage**[1] – *A compact dataset is one in which the raw data is stored in the object header of the dataset. This layout is for very small datasets that can easily fit in the object header. The compact layout can improve storage and access performance for files that have many very tiny datasets. With one I/O access both the header and data values can be read.* When collective metadata reading is enabled, *H5Pset_all_coll_metadata_ops*, and compact storage is used, the metadata will be read by the root process and then broadcasted to the other processes. Note that compact storage has fewer reads than the *read-proc0-and-bcast* solution. Where applicable, the CGNS library was changed to used compact storage instead of contiguous storage. Compact storage datasets have a hard limit of 64KiB. The CGNS implementation uses compact storage datasets by default, but is changed to contiguous storage for the cases of:
    - Datasets greater than 64 KiB,
    - Partial I/O of datasets, meaning IO access to data in a dataset is not exactly the same for all processes. HDF5 can't do partial parallel I/O with compact storage datasets.

CGNS benchmark with various options

In **cgi_read,** at line:

```
for (b=0; b<cg->nbases; b++) if (cgi_read_base(&cg->base[b])) return CG_ERROR;
```

Sky Bridge (SNL) timing:

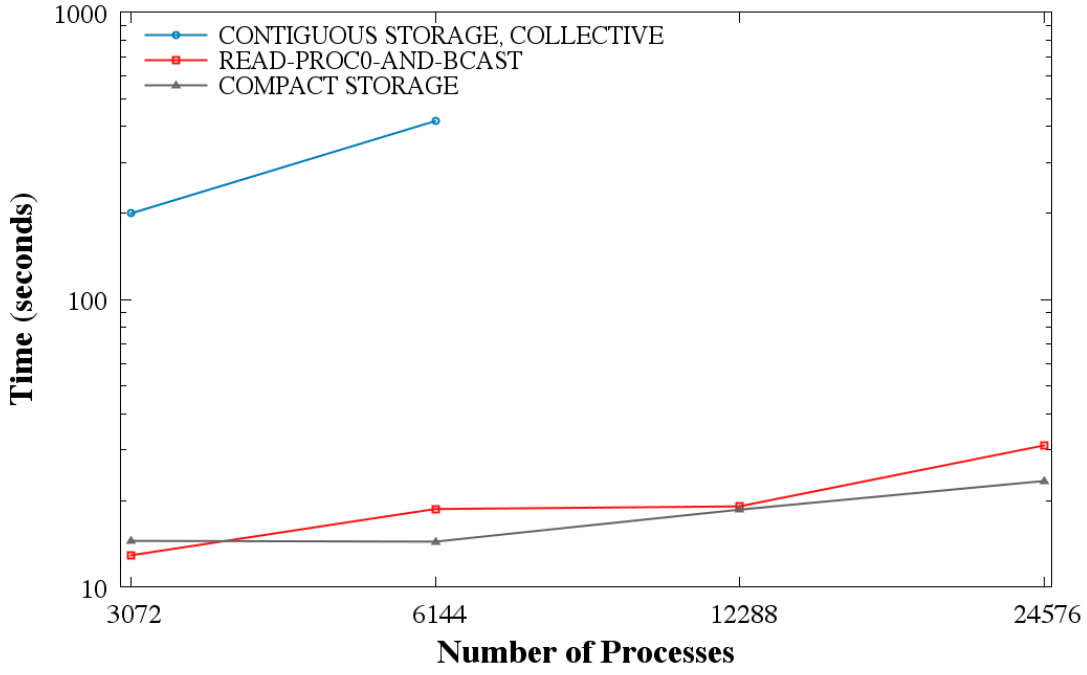
NUMBER OF PROCESSES	Time (seconds)
1	24.4
2048 (H5FD_MPIO_COLLECTIVE)	781
2048 (H5FD_MPIO_PROC0_BCAST)	53.5
2048 (COMPACT STORAGE)	**46.9**
4096 (H5FD_MPIO_PROC0_BCAST)	106.1
4096 (COMPACT STORAGE)	**65.1**
8192 (H5FD_MPIO_PROC0_BCAST)	238.3

8192 (COMPACT STORAGE)	**97.4, 109.9**

Edison (LBNL) timing:

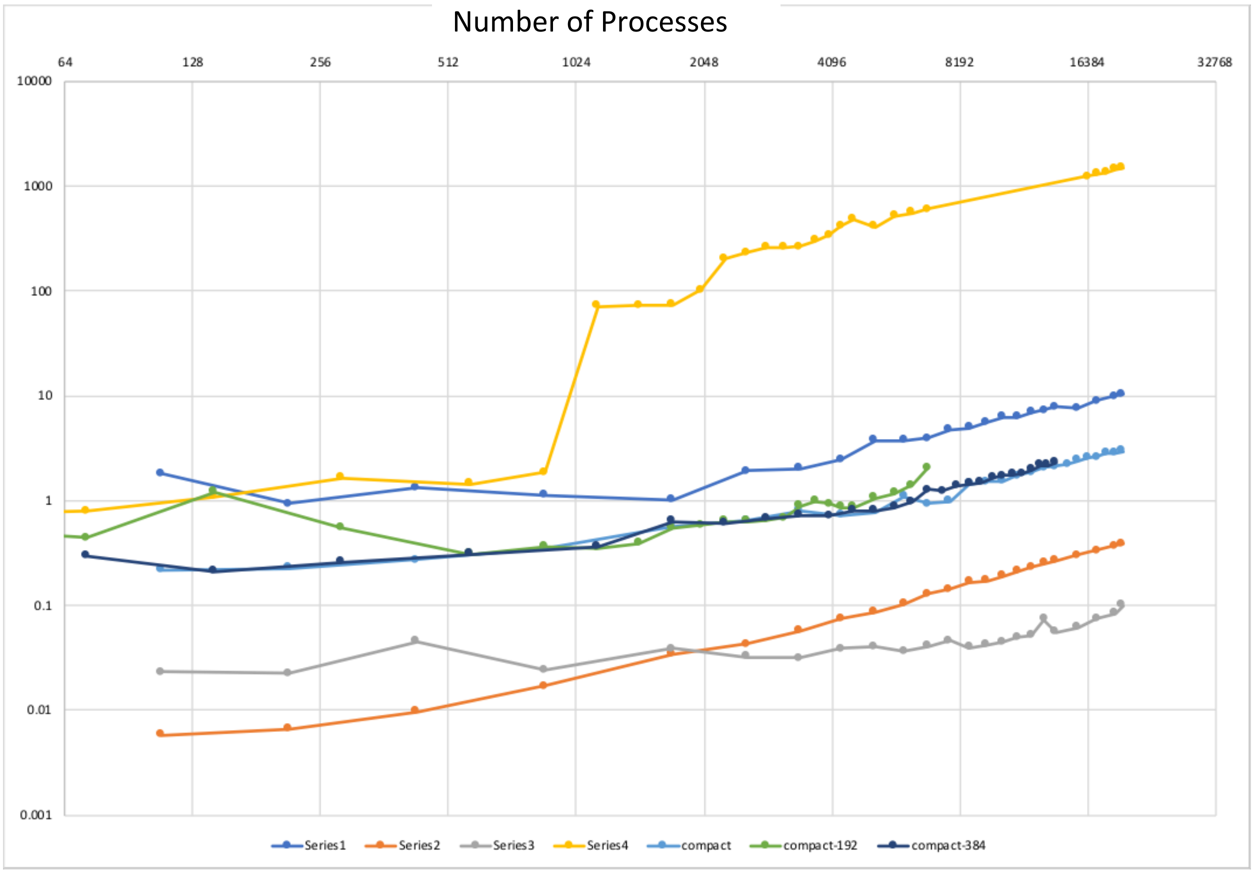
NUMBER OF PROCESSES	**Time**
1	4.9
3072 (H5FD_MPIO_COLLECTIVE)	199.3
3072 (H5FD_MPIO_PROC0_BCAST)	12.9
3072 (COMPACT STORAGE)	**14.5**
6144 (H5FD_MPIO_COLLECTIVE)	417.6
6144 (H5FD_MPIO_PROC0_BCAST)	16.6
6144 (H5FD_MPIO_PROC0_BCAST) **	18.7
6144 (COMPACT STORAGE)	**14.4**
12288 (H5FD_MPIO_PROC0_BCAST) **	19.1 , 26.6
12288 (COMPACT STORAGE)	**18.6**
24576 (H5FD_MPIO_PROC0_BCAST) **	31.1
24576 (COMPACT STORAGE)	**23.4**

** with an extra bcast (for read verification/error checking)



2.1 CGNS application benchmark

A CFD application was used to benchmark CGNS with: (1) compact storage and (2) read-proc0-and-bcast. These results were reported by Greg Sjaardema from Sandia National Laboratories.



- Series 1 is the *read-proc0-and-bcast* solution
- Series 2 is a single MPI_Bcast
- Series 3 uses multiple MPI_Bcast totaling 2 MiB total data 64 bytes at a time (IIRC)
- Series 4 is unmodified CGNS develop
- Compact is using compact storage
- Compact 192 is also using compact storage
- Compact 384 is also using compact storage

The last 3 “compact” curves are just three different batch jobs on 192, 384, and 552 nodes (with 36 core/node). The Series 2 and 3 curves are not related to the CGNS benchmark, but give a qualitative indication on the scaling behavior of MPI_Bcast. Both *read-proc0-and-bcast* and compact storage follow MPI_Bcast’s trend, which makes sense since both methods rely on MPI_Bcast.

2.2 Side investigation: HDF5 small writes of the same values benchmark

2.2.1 HDF5 only test

A benchmark was written to compare writing a small amount of the same data to a file. An integer array of ten elements (i.e., 40 bytes) is written by having:

1. One process creates the file, writes the array, and closes the file.
2. All the processes write the same data to the same location in the file, this is done independently and collectively.

Edison (LBNL)

NUMBER OF PROCESSES	TIME (SECONDS)
1	0.052
3072 (INDEPENDENT)	98.5
3072 (COLLECTIVE)	0.647
6144 (INDEPENDENT)	171.2
6144 (COLLECTIVE)	1.54
12288 (INDEPENDENT)	** timed out after 10 minutes **
12288 (COLLECTIVE)	1.81

Sky Bridge (SNL)

NUMBER OF PROCESSES	TIME (SECONDS)
1	0.012 - 0.020
2048 (INDEPENDENT)	1.5
2048 (COLLECTIVE)	0.17
4096 (INDEPENDENT)	4.7
4096 (COLLECTIVE)	0.49
8192 (INDEPENDENT)	10.1
8192 (COLLECTIVE)	2.4

2.2.2 CGNS test – Only process zero writes

The option to write only with process zero was added to both HDF5 and CGNS for writes of the same data that are less than 4MB.

Edison (LBNL)

NUMBER OF PROCESSES	TIME (SECONDS)
1536 (COLLECTIVE)	2.82
1536 (PROC 0)	2.73
3072 (COLLECTIVE)	3.71
3072 (PROC 0)	3.36

6144 (COLLECTIVE)	4.304
6144 (PROC0)	---

3 Summary of CGNS modifications for scalable read performance

The following changes to the CGNS library should be made:

(1) Switch to using compact datasets where applicable. As far as reading the new compact storage CGNS files, HDF5 handles this switch automatically, i.e. the same CGNS code paths are used to treat both compact and contiguous datasets. An older version of CGNS will not have an issue reading compact datasets since they are in HDF5 versions 1.8 onward. However, to get scalable parallel performance, compact storage needs to be used with collective metadata APIs, and those APIs are only available in HDF5 versions 1.10 onward.

(2) Use the *read-proc0-and-bcast* option when compact storage is not used. This addresses the disadvantage that older CGNS files that don't use compact storage would not scale at large process counts. This option also addresses the case where updating historical CGNS files is not feasible. Furthermore, compact storage is limit to datasets that are less than 64 KiB in size, so datasets exceeding this value would need to use the *read-proc0-and-bcast* option.

From an applications perspective, these proposed changes would be transparent. Also, this is not a CGNS file format change; it just changes internally how the HDF5 library stores the data.

4 The h5dread parallel extension proposal

There are currently two overarching implementations: (1) Give the user complete control to specify the read-proc0-and-bcast solution (Sections 4.1-4.4), or (2) handle switching to the read-proc0-and-bcast solution automatically within the HDF5 library (Section 4.5). For case (1), there are primarily three implementation strategies for reading by process zero and broadcasting the resulting buffer, Section 2. The three implementations address compromises between performance and user error protection.

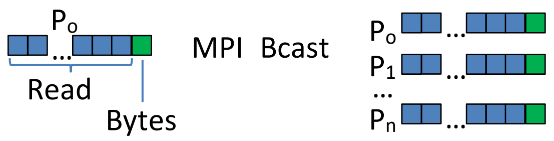
4.1 Implementation Option 1

Implementation 1 is precisely what was prototyped in Section 2. Namely, the lowest process in the MPI communicator reads the data and then broadcasts the data to the remaining processes in the MPI communicator. There are no checks:

1. That all the processors are reading the same data.
2. That the amount of data being read and broadcasted is of a reasonable size.

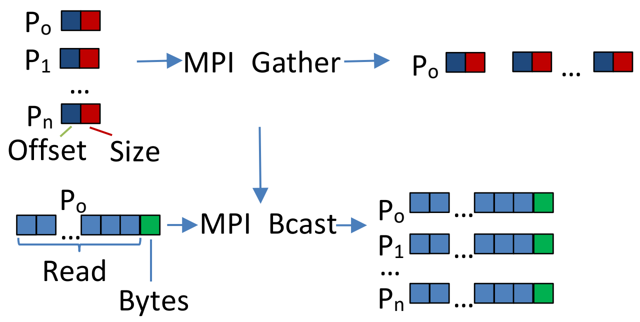
This implementation relies on the user, with no checking in the HDF5 library, to explicitly use this option only when all the processors are reading the same value. This option will always return a read value that is the same for all the processes, regardless of whether this was the actual read usage from the application. Thus, it is possible for a read to "succeed" but return the wrong data.

The main advantage of this option is it has the highest potential in performance since there is a minimum number of broadcasts: (1) the read data, and (2) a one element value being broadcasted. The later (2) is used to determine if the actual number of bytes requested is greater than the bytes read, and if so, the buffer gives zeroes beyond the end of the physical MPI file. This last broadcast (2) can be combined with the read data broadcast (1) to eliminate the need for the extra broadcast.



4.2 Implementation Option 2

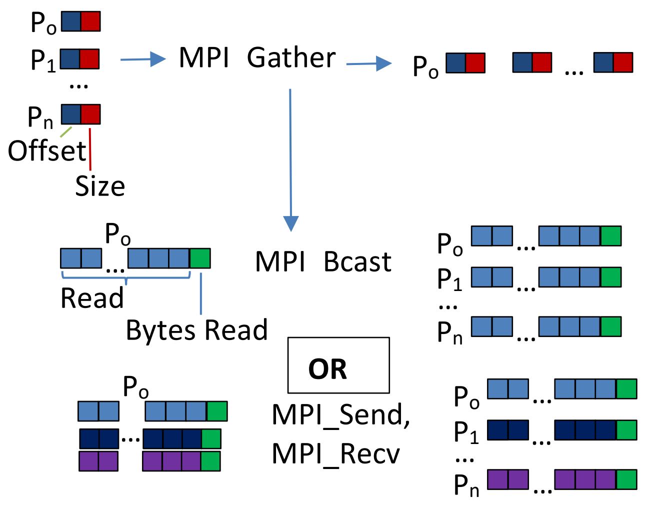
This implementation option adds additional checking to implementation option 1. This implementation checks that the read buffer's offset and size are all the same on all the processors. An additional option would allow the user control for whether or not this check occurs. Overall, this check would add an MPI_Gather to Implementation 1.



If the check is enabled, this option ensures that either the correct data is read by all ranks, or that the read fails. With the check disabled, it is the same as Implementation 1.

4.3 Implementation Option 3

This option ensures that the read on each rank returns the expected data even if the reads are different. Implementation 3 has all the processes send to the lowest processes in the communicator the read buffer's offset and size, as in implementation option 2. If these two parameters all matched, then the gathering process would read the data and broadcast the value to all the matching processes. However, if the parameters do not match, then the gathering processes would read the buffer using the settings from each mismatching process. It would then send the read buffer back to those mismatching processes. This implementation would switch from using an MPI Broadcast to using MPI send and receives when mismatching process parameters were detected, and the performance of using send and receives instead of a broadcast is unknown. In general, this implementation does not impose that all the values read are the same, and in fact can handle cases where the application purposely does not intend the values to be the same. This implementation can also include an option to override all checks and impose all reads return the same value.



4.4 Implementation Issues

All three implementations can fairly easily be implemented into the current HDF5 library. However, options two and three would need to be re-addressed when *Selection I/O* is introduced into HDF5. This new programming model would make the determination of whether the data is the same between processors much more difficult. In that case, the implementation would fail back to implementation one until the feature can be implemented into the new *Object Selections* model.

4.5 Implementation 4: Automatic proc0-read-and-bcast switching in HDF5

The final (and to be completed) implementation uses elements from the previously mentioned implementations. In the HDF5 library, all the processes are reading the same data if they all set the dataset selection to *H5S_ALL*. To determine this requires an MPI_Allreduce. Fortunately, HDF5 already does an MPI_Allreduce for deciding whether to use collective or independent I/O and therefore an additional MPI_Allreduce is not needed in this implementation.

A second criterion in automatically switching to read-proc0-and-bcast is to determine at what dataset size does it becomes faster to do a (1) read-by-all instead of doing a (2) read-proc0-and-bcast. A simple MPI I/O code which does both cases (1) and (2) was developed to determine a possible dataset size to make the switch between the two cases. The code was run on Edison at NERSC on both GPFS and Lustre, Figure 1 and Figure 2 respectively. There is no clear point at which time the read-by-all outperforms the read-proc0-and-bcast. However, as a result of the way HDF5 handles the mpi read (i.e., by the use of derived types), it is not feasible to handle broadcasts of larger than 2GiB. Therefore, a dataset size limit of less than 2GiB for the read-proc0-and-bcast will be used, and datasets greater than 2GiB will revert to using read-by-all. The case of reading an entire, greater than 2GiB dataset, by all the processes should not be a common practice and should be discouraged. For example, on CORI at NERSC, assuming one process per core, reading a 2GiB dataset would consume 64GB of 128GB available memory on a node. Other machine architectures would have similar memory issues in this scenario. Furthermore, the less than 2GiB limitation meets the needs of the current use cases intended for this feature. Alternatively, in cases of greater than 2GiB dataset reads-all is required, the application has the option of implementing the *read-proc0-and-bcast* within the application itself.

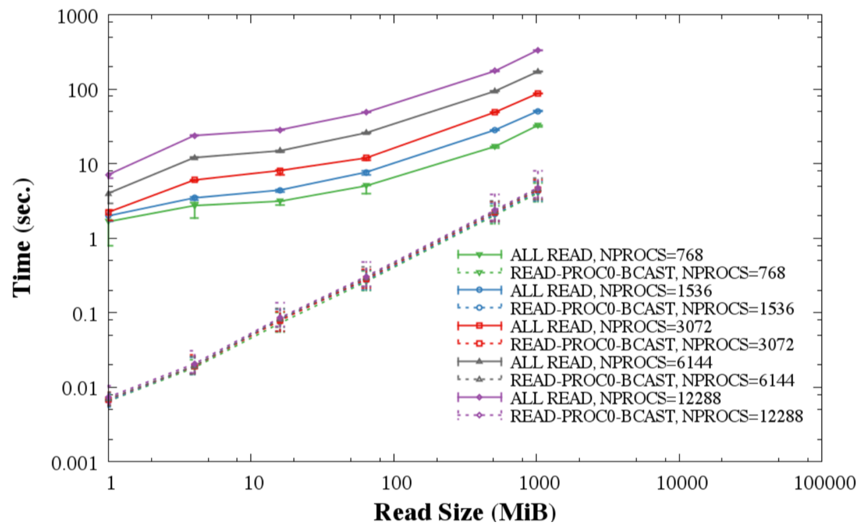


Figure 1 Comparison of read-proc0-and-bcast and read-by-all timing on GPFS (Edison, NERSC).

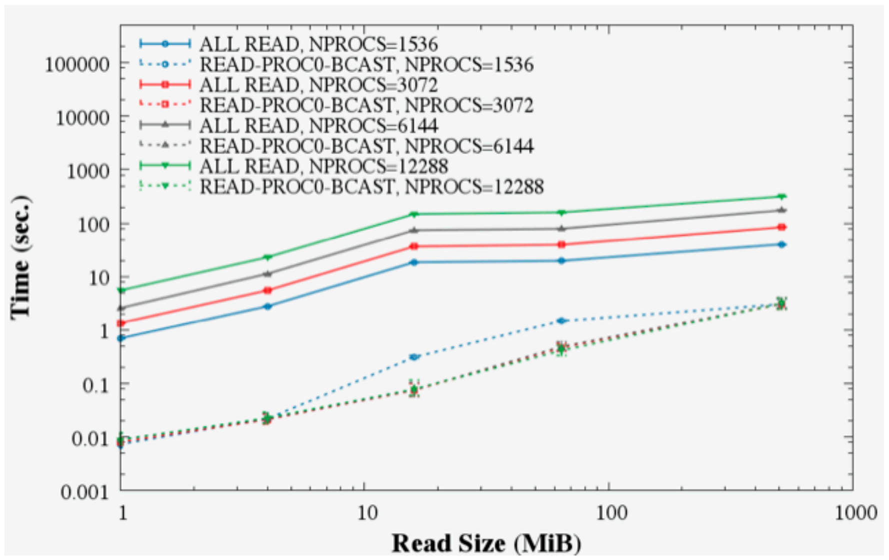


Figure 2 Comparison of *read-proc0-and-bcast* and *read-by-all* timing on Lustre (Edison, NERSC).

5 Conclusion

Guided by the previously highlighted implementations studies, the proposed changes to the CGNS and HDF5 library are:

1. Switch to using compact datasets where applicable in CGNS. As far as reading the new compact storage CGNS files, HDF5 handles this switch automatically, i.e. the same CGNS code paths are used to treat both compact and contiguous datasets. An older version of CGNS will not have an issue reading compact datasets since they are in HDF5 versions 1.8 onward. However, to get scalable parallel performance, compact storage needs to be used with collective metadata APIs, and those APIs are only available in HDF5 versions 1.10 onward.
2. Use a version of HDF5 (1.10.5 onward) where the *read-proc0-and-bcast* is available to handle the case when compact storage is not used. This addresses the disadvantage that older CGNS files that don't use compact storage would not scale at large process counts. This option also addresses the case where updating historical CGNS files are not feasible. Furthermore,

compact storage is limit to datasets that are less than 64 KiB in size, so datasets exceeding this value would need to use the *read-proc0-and-bcast* option.

From an applications perspective, these proposed changes would be transparent. Also, this is not a CGNS file format change; it just changes internally how the HDF5 library stores the data.

References

[1] HDF5 User's Guide, https://support.hdfgroup.org/HDF5/Tutor/layout.html