# HDF5 Collective I/O Performance Investigation

**Jordan Henderson**

This document details a short investigation into collective MPI I/O in HDF5 to verify that the behavior and performance of I/O is reasonable and as expected.

## Revision History

| Version Number | Date | Comments |
|---|---|---|
| v1 | Feb. 21, 2023 | Version 1 drafted. |

## Contents

## 1 Introduction and overview

When improvements were made to HDF5's parallel compression feature in the 1.13.1 release, some quick performance results comparing HDF5 collective MPI I/O read performance before and after were captured using a slightly modified version of IOR that added support for HDF5 data filters. One of these results showed poor performance for baseline HDF5 collective MPI I/O reads (represented by the blue dashed line in Figure 1), so an investigation was made to ensure that HDF5's code for performing collective I/O is reasonable (no extra unnecessary MPI operations, mistaken calls to independent MPI I/O routines, etc.). This document details the results of that investigation and describes the issues uncovered.

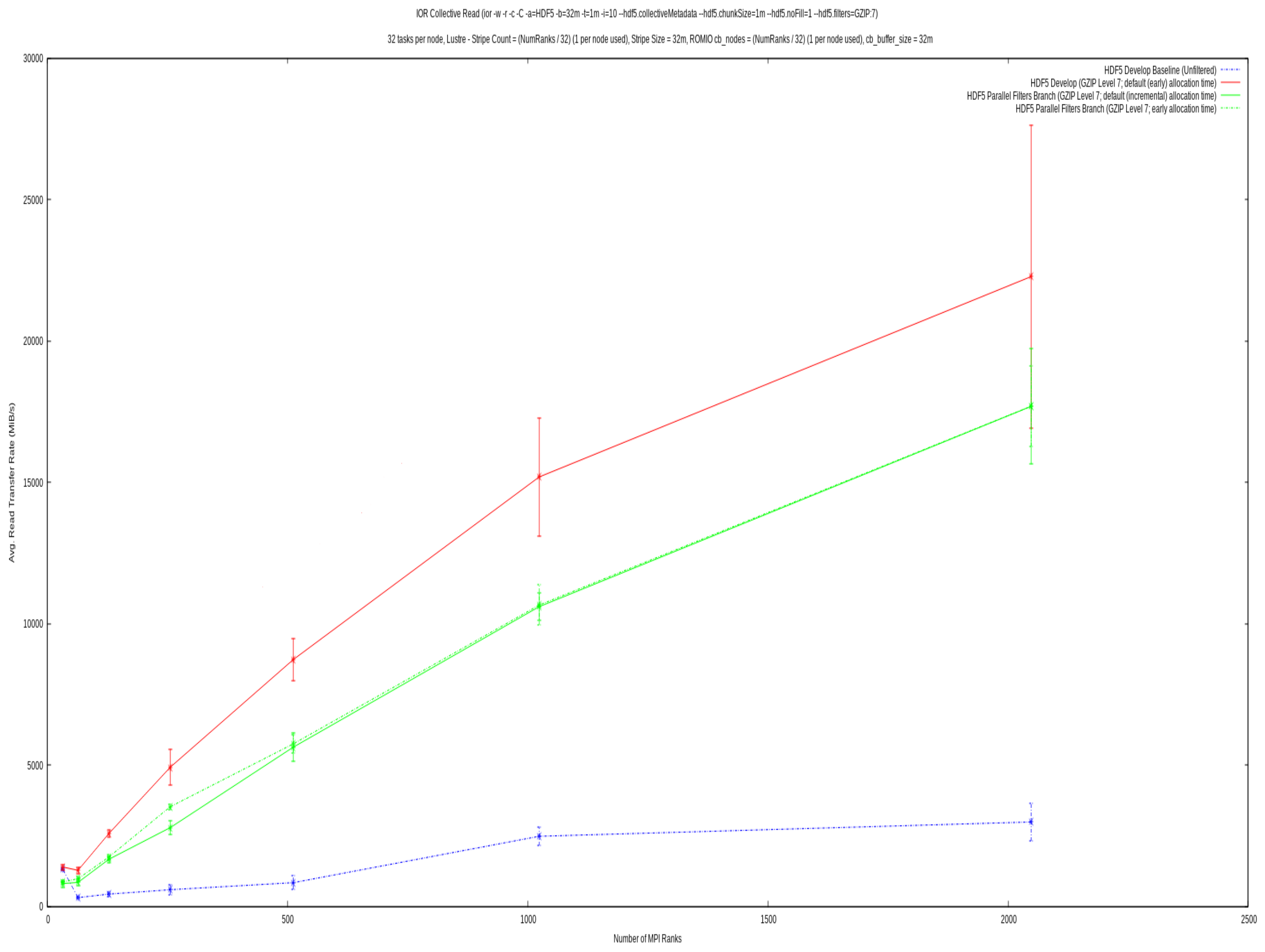


**Figure 1** – IOR HDF5 collective read performance
Command: ior -w -r -c -C -a=HDF5 -b=32m -t=1m -i=10 –hdf5.collectiveMetadata –hdf5.chunkSize=1m –hdf5.noFill
x-axis: number of MPI ranks, y-axis: Avg. read transfer rate in MiB/s

## 2 Investigation Overview

This investigation was carried out on the Vortex machine at Sandia National Labs using IOR and the develop branch of HDF5 at commit 972c883. Experiments were run on the ”/vscratch1” GPFS file system. Unfortunately, an unknown issue prevented experiments from being run at a scale larger than 2 nodes with 88 total MPI ranks. However, the goal of this investigation was primarily to study the number of I/O operations performed by HDF5 as compared to direct MPI I/O, so the tiny application scale used here was still useful. Future investigations should be made at much larger scale when possible.

To begin the investigation, IOR was first run with the MPI I/O API directly by passing the `-a MPIIO` command-line parameter in order to get a baseline for write and read performance to compare against. The backend was run both with and without utilizing MPI file views according to the `--mpiio.useFileView` option. Note that HDF5 parallel I/O matches closest to the case of using MPI file views, but the other case where IOR's MPI I/O backend uses explicit file pointers is an interesting point of comparison. The IOR commands used were as follows:

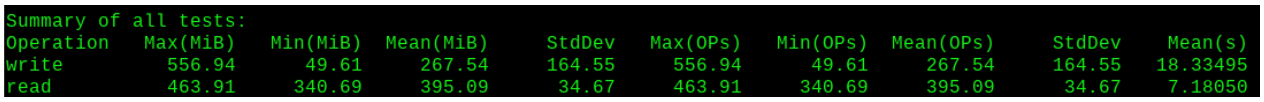
Without MPI file views:

```
$ ior -a MPIIO -w -r -t 1m -b 32m -i 10 -c -C -e
```

With MPI file views:

```
$ ior -a MPIIO -w -r -t 1m -b 32m -i 10 -c -C -e --mpiio.useFileView
```

The IOR results and Darshan profiles for this are shown in Figures 2, 3, 4 and 5.



```
Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev   Max(OPs)   Min(OPs)  Mean(OPs)     StdDev    Mean(s)
write         556.94      49.61     267.54     164.55     556.94      49.61     267.54     164.55   18.33495
read          463.91     340.69     395.09      34.67     463.91     340.69     395.09      34.67    7.18050
```

**Figure 2** – IOR MPI I/O using MPI file views

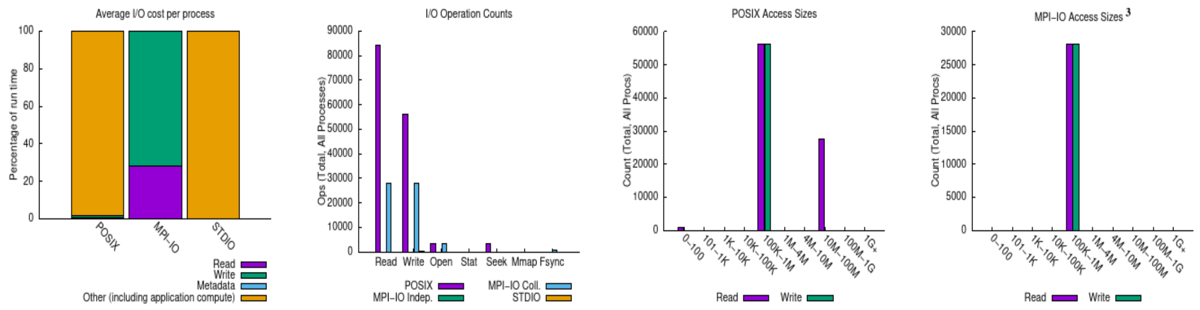
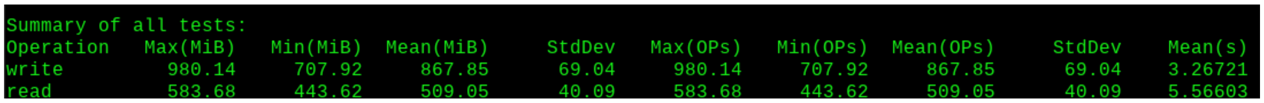


**Figure 3** – Darshan profile for IOR MPI I/O using MPI file views



```
Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev   Max(OPs)   Min(OPs)  Mean(OPs)     StdDev    Mean(s)
write         980.14     707.92     867.85      69.04     980.14     707.92     867.85      69.04    3.26721
read          583.68     443.62     509.05      40.09     583.68     443.62     509.05      40.09    5.56603
```

**Figure 4** – IOR MPI I/O with explicit file pointers

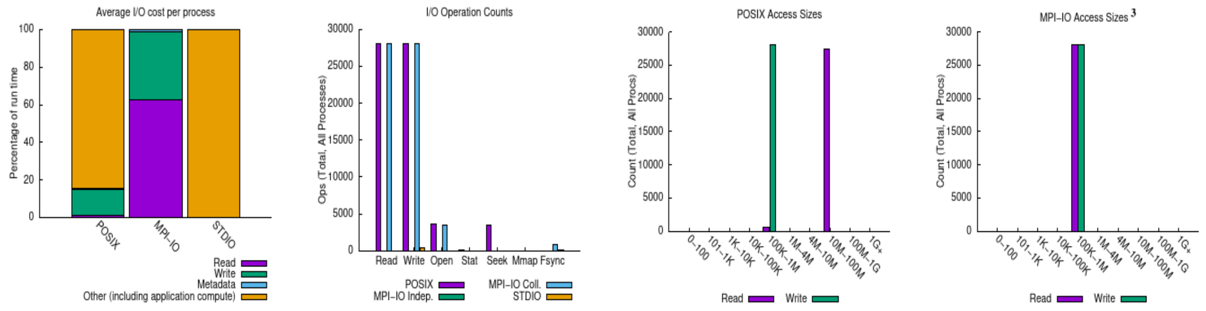


**Figure 5** – Darshan profile for IOR MPI I/O with explicit file pointers

Extracting data from the Darshan profile and matching it to the data shown in Figure 3, it can be seen that the MPI I/O backend to IOR performed 28160 collective MPI I/O reads and writes, as well as 3520 collective file opens, totalled across the 88 available MPI ranks. It can also be seen that all of those reads and writes fell in the 100K-1M range, showing that all the I/O was performed on the specified 1MiB transfer buffer. With a well-behaved HDF5 library, we'd expect to see similar results, with the exception of some minor additional metadata operations.

Next, baselines for HDF5 write and read performance with both contiguous and chunked datasets were obtained by specifying the HDF5 API with the `-a HDF5` command-line parameter for IOR. The IOR commands used were as follows:

Contiguous datasets:

```
$ ior -a HDF5 -w -r -t 1m -b 32m -i 10 -c -C -e --hdf5.noFill
  --hdf5.collectiveMetadata
```

Chunked datasets:

```
$ ior -a HDF5 -w -r -t 1m -b 32m -i 10 -c -C -e --hdf5.noFill
  --hdf5.collectiveMetadata --hdf5.chunkSize=131072
```

The IOR results and Darshan profiles for these are shown in Figures 6, 7, 8, 9. Note that the experiments with HDF5 specify a few additional flags for IOR. The need for these flags is the following:

- `--hdf5.noFill` - Since parallel HDF5 currently forces early file space allocation, the Darshan profiles for these experiments can vary drastically from the MPI I/O profiles due to HDF5 allocating file space and writing fill values out to the dataset. Disabling writing of fill values results in a more fair comparison to the IOR MPI I/O backend.
- `--hdf5.collectiveMetadata` - There are several metadata operations in HDF5 that occur when opening a file (reading the superblock, reading the object header of the file's root group, etc.), opening a dataset (reading of the dataset's object header), writing to a file, etc. By enabling HDF5's collective

metadata reads and collective metadata writes features, we cut down on the number of independent MPI I/O operations performed by all MPI ranks that would otherwise hamper performance and pollute the Darshan profiles captured.

- `--hdf5.chunkSize` - This option simply enables chunking of the HDF5 datasets created by the IOR HDF5 backend. However, note that the value supplied for this argument is currently specified in terms of *dataset elements*, not in terms of bytes.

In the chunked dataset case, IOR was also modified to set the HDF5 library version bounds (both low and high) to `H5F_LIBVER_LATEST` in order to have access to more efficient chunk indexing structures.

```
Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev   Max(OPs)   Min(OPs)  Mean(OPs)     StdDev    Mean(s)
write         626.69     352.57     442.37      77.59     626.69     352.57     442.37      77.59    6.54394
read          423.89     344.63     383.20      20.98     423.89     344.63     383.20      20.98    7.37076
```

**Figure 6** – IOR HDF5 contiguous dataset I/O

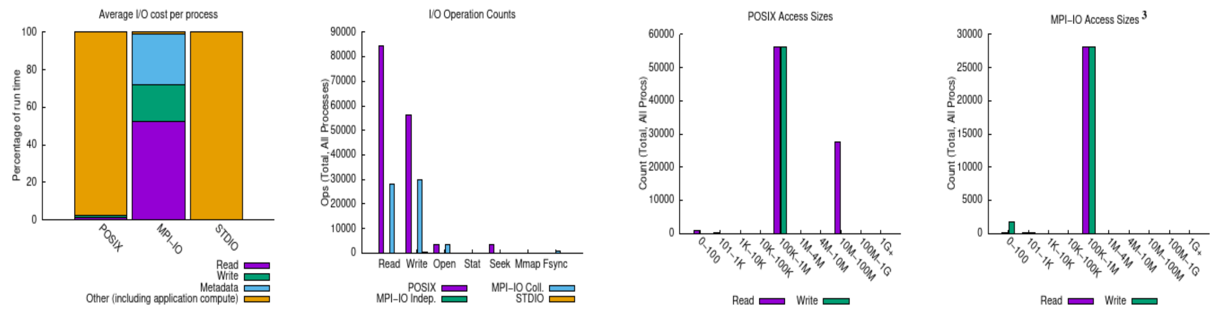


**Figure 7** – Darshan profile for IOR HDF5 contiguous dataset I/O

```
Summary of all tests:
Operation   Max(MiB)   Min(MiB)  Mean(MiB)     StdDev   Max(OPs)   Min(OPs)  Mean(OPs)     StdDev    Mean(s)
write         714.48     399.78     583.15     114.49     714.48     399.78     583.15     114.49    5.04474
read          467.64     334.52     378.07      37.74     467.64     334.52     378.07      37.74    7.51665
```

**Figure 8** – IOR HDF5 chunked dataset I/O

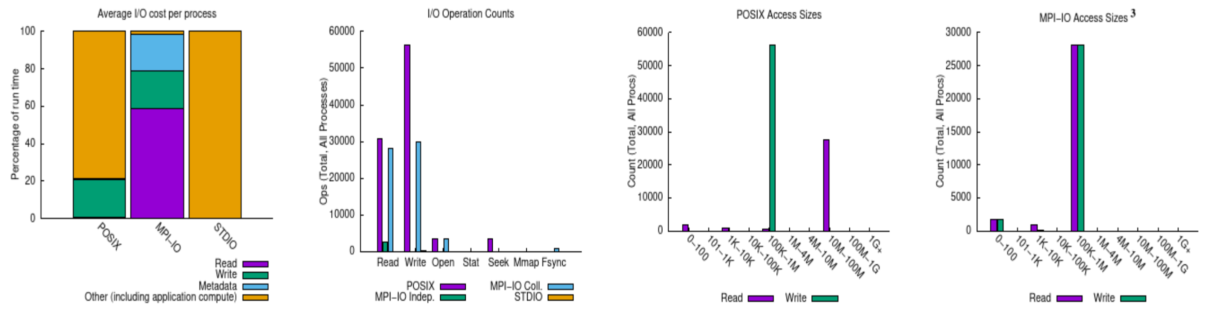


**Figure 9** – Darshan profile for IOR HDF5 chunked dataset I/O

From these figures and the data from the Darshan profiles, we can see that at the MPI I/O level the baseline HDF5 results exhibit an identical 28160 MPI I/O writes and reads in the raw data 100K-1M range, with the contiguous dataset case following extremely closely to the MPI I/O with file views profile. In that case, there are mostly only a few additional calls to collective MPI writes of data in the 0-100 byte range, signalling the activation of HDF5's collective metadata write feature. However, the chunked dataset case varies a little bit in that there are significantly less POSIX reads in the 100K-1M range and slightly more reads in the 0-100 byte and 1K-10K ranges which appear to all be independent reads based on the Darshan graph. Investigation revealed that the 0-100 byte range reads are related to looking up chunk metadata, while the 1K-10K reads are accidental independent reads of a dataset's chunk index, described in section 3.

As a last note, typically one would make sure to set the alignment of objects in the HDF5 file using the `H5Pset_alignment` API routine on an HDF5 File Access Property List to ensure that objects are well-aligned with the parallel file system's data striping. In this case though, neither setting the alignment to be equal to the chosen chunk size and IOR transfer size of 1MiB nor setting it to the Darshan-detected file alignment size of 4MiB appeared to have any noticeable affect on performance or the generated Darshan profiles.

## 3 Observed independent reads

During the investigation of collective I/O performance, several independent MPI I/O reads appeared in the captured Darshan profiles when performing I/O on chunked datasets, as shown in Figure 9.

Enabling debugging in the MPI I/O VFD and investigating further showed that these independent reads were primarily occurring in two places: lookup of a dataset's chunks and reading of a dataset's chunk index. Other independent reads occurred in various places, such as during reading of the file's superblock, reading of object headers, etc., but these independent reads aren't generally of interest because they only occurred on MPI rank 0, signalling that the collective metadata reads feature was operating properly in those places.

### 3.1 Chunk lookup

When HDF5 looks up chunks for a chunked dataset, it is expected that the library will perform those lookups with independent MPI I/O operations, even when collective metadata reads are enabled. The relevant section of the library's code contains the following comment:

```
/* Disable collective metadata read for chunk indexes as it is
 * highly unlikely that users would read the same chunks from all
 * processes.
 */
```

Typically, this shouldn't be much of an issue for parallel HDF5 applications as different processes should usually be reading from different (or mostly different) chunks with no or minimal overlap. When combined with an optimal file striping strategy on a parallel file system, this should generally spread the I/O across separate data stripes and thus multiple storage targets. If the application actually is reading the same chunks on all ranks, it could likely be expected that a collective read of those chunks would be more efficient, but, as the previous comment notes, this isn't very likely and the argument could be made that this is a deficiency of the application that should be corrected (perhaps by some rank-0-broadcast strategy at the application level).

It may be the case (especially with modern optimizations in MPI implementations) that a collective read/lookup of all the processes' chunks would perform better than the several independent reads, even after factoring in the collective communication overhead that would be required for coordinating the collective read between ranks. A future investigation should be made to verify this so that the edge case of turning off collective metadata reads for chunk lookups can potentially be eliminated and optimized.

### 3.2 Chunk index read issue

With debugging enabled in the MPI I/O VFD, it was noted that several small independent reads of B-tree data were occurring during IOR's dataset read iterations. In a typical `H5Dopen` → `H5Dread` operation, the library's first interaction with a dataset's chunk index usually occurs within the chunk lookup operation mentioned in the previous section. During that lookup operation, the library will determine that the dataset's chunk index hasn't been read in and opened yet, so it will attempt to do so before looking up the dataset's chunks. Unfortunately, since the chunk lookup operation will have temporarily disabled collective metadata reads, the initial reading of the chunk index will become an independent read operation that is performed by all MPI ranks. This issue doesn't appear during a `H5Dcreate` → `H5Dwrite` operation because the dataset's chunk index is created in `H5Dcreate` and therefore doesn't need to be read in.

As this will entail a small independent read of the same file location across all ranks that were involved in opening the dataset, it's possible that this can prove to be a performance issue at large scale by causing a "metadata storm". Of course, results may vary depending on the machine and capabilities of the underlying file system. It's worth noting that IOR measures the HDF5 dataset open time as part of its read operation timing and performance measurement, so results could likely become skewed if those small metadata reads become a problem with increasing MPI rank counts. Anecdotally, adding a workaround to the HDF5 library code to sidestep this issue and read the dataset's chunk index collectively appeared to improve the consistency in measured read performance in IOR, but further investigation at much larger scale will be needed to see exactly how much of a problem these independent reads are.

The fix for this issue will likely be to add a new chunk index routine to check if the index has been read and then modify the chunk lookup routine to make use of this new routine to determine if the chunk index needs to be collectively read in (if collective metadata reads are enabled) before disabling collective metadata reads. While the chunk lookup routine is called often, adding a check of whether the chunk index has been read shouldn't incur any noticeable overhead since it's a simple check of a pointer for all the current chunk indexing methods.

### 3.3 Object header and B-tree reads

In an attempt to see if it was possible to further minimize the metadata read operations being performed during object header lookups, reads of B-tree information, etc., a small experiment comparing different metadata block sizes, as set with the H5Pset_meta_block_size API routine, was performed. However, the experiments performed in this investigation were at small enough scale that the library's default metadata block size of 2048 bytes already appeared to be optimal.

## 4 Performance of collective metadata writes

During this investigation, it was noted that performance could trail behind IOR's MPI I/O backend slightly when collective metadata writes are enabled (the default in IOR when the `--hdf5.collectiveMetadata` option is enabled) and the metadata generated isn't very large. The description of HDF5's collective metadata write API mentions that it's recommended to use the feature when the size of metadata created is large, but it also may not be easy for the application/library to predict this case ahead of time. Based on this, it may be an interesting idea to introduce a new or revised collective metadata write API to HDF5 that accepts a threshold value below which metadata will be written independently and above (or equal to) which metadata will be written collectively. In this manner, a parallel HDF5 application may be able to further tune performance in an adaptive way.

## 5 Conclusion

The investigation described in this document finds that there are, as of the time of writing, no direct issues with the collective MPI I/O pathways in HDF5 for raw dataset data. However, there exist a few issues with how HDF5 deals with file metadata that can potentially be optimized for large-scale MPI applications. As the number of I/O operations and I/O performance of HDF5 parallel I/O match closely to their counterparts for the MPI I/O backend in IOR, the tentative conclusion is that the original issue prompting this investigation was likely due to an issue with how performance testing was setup and performed on that particular machine and its parallel file system. Further investigation on that machine at large scale will be needed to determine why there existed issues with parallel HDF5 I/O performance.