File Open, Close, and Flush Performance Issues in HDF5

Scot Breitenfeld
John Mainzer
Richard Warren
02/19/18

1 Introduction

Historically, the parallel version of the HDF5 library has suffered from performance issues on file open, close, and flush, which have typically been related to small metadata I/O issues. While the small metadata I/O issue has been addressed in the past, initially by distributing metadata writes across processes, and subsequently by writing metadata collectively on file close and flush and supporting collective metadata reads in some cases, the problem has re-appeared as the typical number of processes in HPC computations has increased.

In this paper, we outline the results of the latest efforts to address the metadata I/O issue and its effects on HDF5 file open, close, and flush performance.

As shall be seen, the problems encountered and our solutions to them have ranged from trivial to fundamental. In particular, our work on the Avoid Truncate Issue suggests a way to avoid the small metadata I/O issue entirely, albeit at the cost of increased memory footprint.

2 Slow File Open

While investigating other issues, we noticed that while the superblock read on file open is collective when collective metadata reads are enabled, all processes independently search for the superblock location – which at a minimum means that all processes independently read the first eight bytes of the file.

As this is an obvious performance bottleneck on file open for large computations, and the fix is simple and highly localized, we modified the HDF5 library so that only process 0 searches for the superblock, and broadcasts its location to all other processes. At the time, we had not received recent complaints about slow file open, and thus we did not validate the fix with before and after performance studies.

We subsequently received a second hand report of slow file open with very large numbers of processes. While we don't know the details on this issue, we expect that the above fix will improve matters significantly, particularly when combined with collective metadata reads.

3 Duplicate Metadata Writes on Flush and Close

A file close slowdown with collective metadata writes enabled was observed after the HDF5 1.10.1 release. This was tracked to duplicate metadata writes on close or flush (once collectively, and again independently), which in turn was tracked to a merge error in the preparation of HDF5 1.10.1. The fix was trivial, and will be included in HDF5 1.10.2.

Unfortunately, writing a regression test for this issue is difficult. However, as the HDF Group is adding parallel performance regression tests, we may hope that parallel performance bugs of this type will be caught prior to release in the future.

4 Long Close Times for Files Opened R/W but not modified

As shall be seen in the Avoid Truncate Bug section below, we have observed cases in which file close is slower for files opened R/W but not written to than for files opened R/W and written to. From first principles, file close in the former case should be faster, and thus we presume that this is bug that should be addressed – particularly as the delay is significant (several seconds in some cases).

To date, we have not had the resources to investigate the issue.

5 Avoid Truncate Issue

The "avoid truncate" bug dates back to at least 2009, when we learned of a case in which a computation with either 512 or 2048 processes was spending most of its time truncating the HDF5 file in preparation for a file close or a flush. More generally, similar delays on HDF5 file close and/or flush have been observed on some, but not all, Lustre file systems.

Initial attempts to resolve this issue seem to have assumed that the fundamental problem is that MPI_File_set_size() is slow on some Lustre file systems, and thus that the appropriate solution is to avoid calls to MPI_File_set_size() at file close and/or flush. The solutions investigated were either to modify the HDF5 file format to no longer require that end of allocation (EOA) and end of file (EOF) match at file open or to call the POSIX truncate() function from rank 0 after file close to force the EOF to match the EOA.

As shall be seen on various systems (Table 1), this presumption that MPI_File_set_size() is the problem came into question as the result of the failure of the "call POSIX truncate() from rank 0" approach to yield the expected results.

Host Name	Site Location	Machine Type	File System
Cori	DOE – NERSC	Cray XC40	Lustre
Excalibur	DOD - HPCMP	Cray XC40	Lustre

Table 1 Computer systems used for the experiments.

The Problem and its Cause

Figure 1 illustrates the "avoid truncate" problem by comparing the file close time of an un-altered version of the HDF5 library, with that of a version in which the call to MPI_File_set_size() is omitted unconditionally. On the face of it, it makes the solution obvious – just avoid the call to MPI_File_set_size(), and the problem goes away.

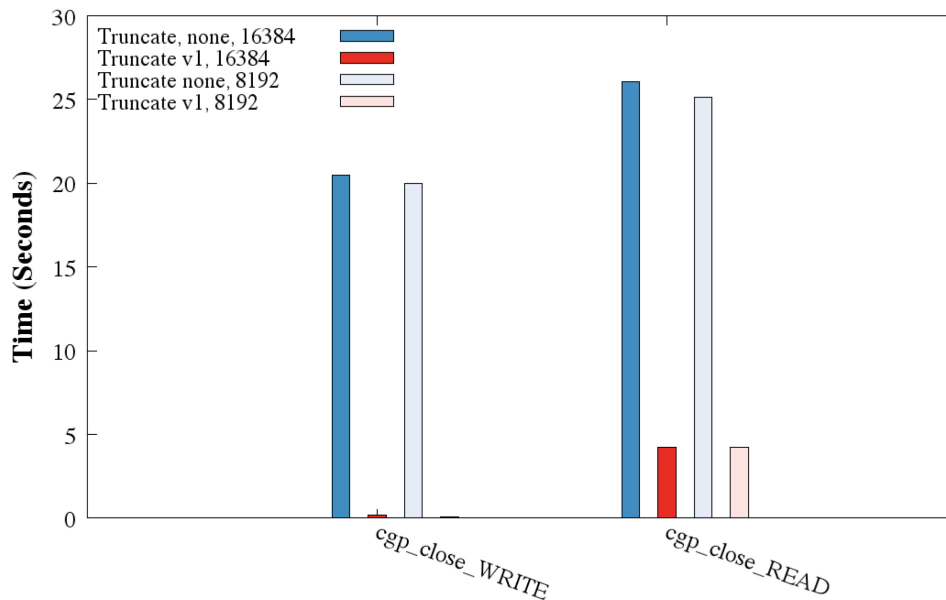

Figure 1 File close times for a file opened read/write (cgp_close_WRITE) and read only (cgp_close_READ) for an un-altered version of the HDF5 library (no patch) and a version in which the call to MPI_File_set_size() is omitted unconditionally (v1). Note the long close times for files opened R/O as discussed above.

In cases in which datasets are allocated, completely defined, never deleted, and the number of ranks is a power of two, the call to MPI_File_set_size() is usually unnecessary, as the EOA and EOF will usually match. However, in the more general case, we must truncate the file on close so that EOF equals EOA, as the HDF5 library checks this on file open, and refuses to open the files when this condition does not hold on the presumption that the file has been corrupted.

One way around this issue is to store both the EOA and the EOF, and simply verify that the EOF has the expected value on file open. However, this option requires a file format change (usually done only on major releases of the HDF5 library) and introduces some interesting forward compatibility issues, as, without a conversion utility, earlier versions of the HDF5 library would not be able to open files R/W.

In an attempt to sidestep this issue, we investigated the possibility of performing the truncate with a POSIX truncate() call from process 0 after file close. The results were not encouraging, Table 2. More importantly, the fact that the POSIX truncate was taking longer than the MPI truncate caused us to begin to doubt that MPI_File_set_size() was the fundamental problem.

No truncate	**0.083 sec**.
MPI truncate	45.0 sec.
POSIX truncate	72.1 sec.

Table 2 File close time for 3,000 processes on Cori using collective metadata writes.

Acting on the guess that the issue might be related to the many small metadata writes that HDF5 performs on file close, we wrote a simple MPI program in which each process writes one or more well-aligned blocks of data, and then closes the file either:

1. without calling MPI_File_set_size(),
2. truncating the file to its current length via a collective call to MPI_File_set_size() just before file close, or
3. truncating the file to its current length via a call to truncate() after file close.

As there was almost no difference between these three cases, Table 3, the results of this benchmark are consistent with the hypothesis that the problem is not with MPI_File_set_size(), but instead with the HDF5 metadata layout.

Case No.	**Time (seconds)**
1	0.00693
2	0.00890
3	0.00963

Table 3 Results of MPI benchmark on Excalibur for 8192 processes in which each process writes one or more well aligned blocks collectively. Timings are for file close with either: (1) no truncate, (2) truncate to current length via MPI_File_set_size() just before file close, or (3) POSIX truncate() by process 0 just after file close.

At this point it became clear that the “slow H5F_File_set_size() on some Lustre installations” hypothesis was probably false – and that if we wanted to solve the problem, we needed to determine the fundamental cause.

To this end, we entered into a lengthy email discussion with members of the Lustre development team at Intel. After much back and forth (and generous patience on the part of the Lustre team in general, and Andreas Dilger in particular), we arrived

at the following summary of the likely cause of the problem (quoting from our final email exchange with Andreas).

1) *HDF5's current practice of scattering small (i.e., typically less than a few KB) pieces of metadata in small bunches throughout the file, and then writing dirty metadata in a single collective write just before close results in excessive lock contention.*

   Further, since the typical total size of this collective metadata write is < 1 MB, I gather that there isn't much we can do to optimize it further without changing our metadata layout on file. If we had more metadata, I gather that it might be useful to bin the metadata by the OST it resides on, and then write the metadata with one process writing the contents of each bin. However, this is questionable, due to the typically very low density of metadata in HDF5 files.

2) *The MPI_File_set_size() call that is made shortly after the above collective metadata write typically cannot execute until it obtains locks on all OSTs. This requires the call to wait until the above contention is resolved -- hence the delays we have seen on our truncate calls at file close.*

3) *Aggregating the metadata into page aligned pages of size some multiple of 1 MB, and then performing I/O on the pages instead of individual pieces of metadata should be a great win, and should avoid the lock contention mentioned in 1).*

   The aggregation into pages should still be useful in cases where we cannot use page buffering (most likely due to memory footprint), particularly if we use your new progressive file layout feature and pre-allocate space for metadata at the beginning of the file.

While the above seemed consistent with our observations to date, we further discussed the problem and the above diagnosis with some people at SC17 – most notably current and former members of the Panasas development team, and Rob Latham of ANL / MPICH. The consensus was that the above diagnosis is likely correct. However, it should be noted that as of our last communication on the subject, Quincey Koziol was still of the opinion that the problem was slow implementations of truncate on Lustre.

As implementing the changes to the HDF5 library suggested above will not be trivial, we also wrote another pair of MPI test programs comparing the following scenarios:

1) Large chunks of data are interspersed with randomly scattered small clusters of small (i.e. < 5 KiB) pieces of data totaling 1 – 2 MiB. Large data is written in several collective writes. Small data is also written collectively just before

file close, with at most one piece of small data per process. File is truncated to its current size at close via MPI_File_set_size()

2) Small pieces of metadata are collected into one or two well aligned 1 MiB blocks. Large data is written as before. Small data is written collectively at file close with one or two processes each writing a 1 MiB block, with all other processes participating in the write, but not contributing. File is truncated to its current size just prior to close via MPI_File_set_size().

Addendum 2/11/18

Initial results of the above two benchmarks on Cori were not what we expected. As we received a stop work order on ECP shortly thereafter, we were unable to investigate further.

The obvious possibilities are that the first benchmark does not adequately replicate the behavior of HDF5, that our diagnosis is wrong, or that it only applies to some machines.

Solutions

Given the diagnosis outlined above, two solutions present themselves – a stopgap solution, and a general solution of the issue.

The stopgap is to query the file system on file flush or close, and skip the call to MPI_File_set_size() if the file system's EOF and the HDF5 library's EOA agree. This fix is easy, so we implemented a prototype for testing. While there is some overhead in querying the file system and broadcasting the results to all ranks, initial benchmarks are encouraging, Table 4. However, since we don't currently have access to a Lustre file system that shows the problem well, we are holding off on merging this solution into the HDF5 library proper until we can validate it on other Lustre systems.

	FILE CREATE: H5Fclose	**FILE OPEN RW: H5Fclose**
No Fix:	0.7 Sec	22.12 Sec
Skip Truncate unconditionally	0.43 Sec	15.5 Sec
Skip Truncate when possible	0.5 Sec	14.0 Sec

Table 4 Initial results of benchmark of stopgap fix on Cori with 2048 processes. (Scot – some of these numbers look odd – we should discuss). We should redo these runs –Scot—

Addendum: 2/19/18: We have made this patch available for testing. So far, we have one positive result – details to follow.

The general solution is to re-work HDF5 metadata layout and I/O to avoid the lock contention discussed above.

In HDF5 1.10.1, we release paged metadata and small raw data allocation in serial and parallel, and page buffering in the serial case only. Thus, we already have the ability to aggregate metadata in pages of user-configurable size. Page buffering was not enabled in parallel as cache coherency issues make it difficult to support this feature for raw data. This is not an issue for metadata, as metadata is already consistent across all ranks, and thus no fundamental issues prevent us from using it for metadata in the parallel case. However, the thinking at the time was that collective metadata I/O made the point moot.

While there are no fundamental or novel problems with enabling page buffering for metadata in parallel, this is not to say that the problem is trivial. Implementation will require significant reworking of the page buffer to integrate it into HDF5 parallel metadata I/O management, to maintain coherency across processes, and possibly to support collective metadata page reads. Further, since this solution to the avoid truncate bug will likely increase HDF5 library memory footprint; we must preserve our current metadata I/O management scheme as an option where severe constraints on memory footprint make the page buffer approach impractical. While design work will be required to come up with firm estimates, my impression at present is that it will be about three months of work.

In considering the above general solution to the avoid truncate bug, it should be noted that it points the way to solving the small metadata I/O problem in HDF5 completely.

Specifically, if we modify the file format to allow us to maintain a list of all pages of metadata in a superblock extension message, and if the metadata is small enough that it is practical to keep it all in the page buffer, on file open, process 0 could load all metadata pages and broadcast them to all other processes, thus avoiding all reads of individual pieces of metadata.

Similarly, all dirty metadata could be written to file in a single collective write of dirty metadata pages on close or flush, thus avoiding all small metadata writes as well.

Further, since all metadata cache misses would be satisfied from the page buffer in this scenario, we could restrict metadata cache size more aggressively without incurring unacceptable performance penalties. As metadata in the page buffer is in on disk format which requires anywhere from a third to a tenth of the RAM required for the same piece of metadata in the metadata cache where it is unpacked for

immediate use, the memory needed for the page buffer could be at least partially offset by reduced metadata cache size.

In the common scenario in which the metadata in the HDF5 file fits into at most a few 1 MiB pages, metadata I/O is restricted to file open, close, and flush, and is aggregated into pages sized to match the underlying file system page size. This should yield a large win, as metadata I/O will be indistinguishable from raw data I/O, and we will be able to avoid the overhead of the current sync point scheme for flushing dirty metadata during the course of the computation.

In cases where the metadata is too large to be retained in the page buffer for the duration of the computation, we will have to retain the sync point scheme to allow dirty pages of metadata to be flushed without the risk of message from the past/future bugs. However, these writes will be collective writes of pages sized to the underlying file system, and thus should avoid lock contention issues and the resulting delays. While we will not be able to avoid independent metadata page reads completely, we should be able to use collective page reads in many cases.