RFC: Enabling a Strict Consistency Semantics Model in Parallel HDF5

Mohamad Chaarawi

Quincey Koziol

Consistency semantics define the outcome of multiple I/O accesses to the same file. Any I/O library defines these rules to let users know what to expect when accessing data in the file. A process that writes data to the file needs to be aware when those changes made are actually visible to itself (if it tries to read back that data) or to other processes who are concurrently accessing the file independently or collectively. This RFC describes the consistency semantics model for the Parallel HDF5 library when it comes to metadata and raw data access, and outlines some issues with that model and approaches to overcome them.

Introduction

A parallel file system allows striping of data over multiple servers to achieve high performance. Parallel HDF5 provides users with an option to have multiple processes perform I/O to an HDF5 file at the same time. It is not a requirement that the HDF5 file accessed by the library resides on a parallel file system; however, it only makes sense for parallel access to a file to occur on a parallel file system, otherwise the parallel access will introduce unnecessary contention on the file system.

The HDF5 library makes use of virtual file drivers to take care of the lower level I/O accesses to an HDF5 file. Each of those drivers has certain consistency rules that define the outcome of accessing an HDF5 file. A process that writes data to an HDF5 file needs to be aware of when those changes made are actually visible to itself (if it tries to read back that data) or to other processes that may be concurrently accessing the HDF5 file independently or collectively.

There are several scenarios that arise from parallel access to a parallel file system using the MPI-I/O file driver, but the most common cases include:

1. Each process accesses its own separate file: There are no conflicts in this case, which means data written by a process is visible immediately to that process.
2. All processes open the file(s) in *read-only* mode: All processes will read exactly what is in the file.
3. Multiple processes access the same file with read-write operations, which is the scenario discussed in this document.

Depending on the file driver being used, the consistency semantics vary. For example, the MPI-POSIX driver guarantees sequential consistency because POSIX I/O provides that guarantee. Consider a scenario with two processes accessing data at the same location in the same file:

Process 0	Process 1
1) write ()	1)
2) MPI_Barrier()	2) MPI_Barrier()
3)	3) read ()

POSIX I/O guarantees that Process 1 will read what Process 0 has written, because of the atomicity of the read and write calls and the synchronization using the barrier that ensures that Process 1 will call the read function after Process 0 is finished with the write function.

On the other hand, the MPI-I/O driver does not give that guarantee. If we take the example above and substitute the write() and read() calls with MPI_File_write_at() and MPI_File_read_at() respectively, MPI-I/O with its default settings does not guarantee sequential consistency. In the example, Process 1 is not guaranteed that it will read what Process 0 writes. In fact, the read might occur before, after, or even during the write operation. MPI-I/O does, however, provide different options to ensure sequential consistency or user-imposed consistency, which we will discuss further in the next sections. For a full discussion on MPI-I/O consistency semantics, please refer to the MPI-2.2 standard (http://www.mpi-forum.org/docs/ section 13.6, page 437, line 44).

This RFC describes the consistency semantics model for the Parallel HDF5 library when it comes to metadata and raw data access to a shared file among processes using the MPI-I/O file driver, and outlines some issues with that model and approaches to overcome those issues.

Motivation

MPI provides the user with options to impose different levels of consistency. If the user sets the atomicity flag to true, all MPI access functions are guaranteed to execute atomically. For example:

Process 0	Process 1
1) MPI_File_set_atomicity (fh, TRUE)	1) MPI_File_set_atomicity (fh, TRUE)
2) MPI_File_write_at ()	2) MPI_File_read_at ()

The read by process 1 will read the entire data before the write by process 0 occurs or after (not during).

If the user wishes to impose an ordering between the read and write operations, a barrier (or any other method to ensure ordering) can be used. For example:

Process 0	Process 1
1) MPI_File_set_atomicity (fh, TRUE)	1) MPI_File_set_atomicity (fh, TRUE)

2) MPI_File_write_at()	2)
3) MPI_Barrier()	3) MPI_Barrier()
4)	4) MPI_File_read_at ()

This will ensure that Process 1 reads what Process 0 has written (the read happens logically after the write).

There are other ways for the user to impose this level of consistency on file access operations:

1) Close the file after the write operation (with all processes) and re-open it, then issue the read operation.
2) Ensure that no write sequence on any process is concurrent with a read or write sequence on any other process, as shown in the next table. This is a general characterization of the sync-barrier-sync construct, and is a more complicated way to impose a strict consistency semantics model. Please refer to the MPI standard for a more detailed discussion on this method.

Process 0	Process 1
1) MPI_File_write_at()	1)
2) MPI_File_sync()	2) MPI_File_sync()
3) MPI_Barrier()	3) MPI_Barrier()
4) MPI_File_sync()	4) MPI_File_sync()
5)	5) MPI_File_read_at ()

The previous sequence will ensure that process 1 reads what process 0 has written.

PHDF5 Metadata Consistency Semantics

HDF5 files can contain several types of metadata. For a discussion of these different types of metadata, please refer to the "*HDF5 Metadata*" document in the Advanced Topics section of the User Guide (http://www.hdfgroup.org/HDF5/doc/Advanced/HDF5_Metadata/index.html). In the parallel version of the HDF5 library, there are several constraints that limit the ability to update HDF5 metadata on disk. To handle synchronization issues, all HDF5 operations that could potentially modify the metadata in the HDF5 file are required to be collective. A list of those routines is available in the HDF5 reference manual (http://www.hdfgroup.org/HDF5/doc/RM/CollectiveCalls.html).

Since all operations that modify metadata in the HDF5 file are collective, all processes will have the same dirty metadata entries in their cache (i.e., metadata that is inconsistent with what is on disk). There is no communication necessary to mark metadata as dirty in a process's cache, as all the calls are collective, and all processes see the same stream of changes to metadata. Operations that just read metadata (for example, H5Dopen) are not required to be collective, and so processes are allowed to have different clean metadata entires (i.e., metadata that is in sync with what is on disk). Internally, the metadata cache running on process 0 of the file communicator is responsible for managing changes to the metadata in the HDF5 file. All the other caches must retain dirty metadata until the process 0 cache tells them that the metadata is clean (i.e., on disk). The actual flush of metadata entries to disk is currently implemented in two ways:

A – Single Process (Process 0) writes:
1) perform a barrier
2) process 0 writes each piece of metadata individually
3) process 0 broadcasts the list of flushed metadata to other processes
4) all processes mark the flushed metadata entries as clean in their cache

B – Distributed:
1) perform a barrier
2) process 0 broadcasts the list of metadata to flush to other processes
3) each process determines which pieces of metadata it should write
4) each process writes each piece of metadata independently
5) each process participates in an exchange of whether any errors occurred
6) all processes mark all flushed metadata entries as clean in their cache

Please refer to the *"Metadata Caching in HDF5"* entry in the Advanced Topics section of the HDF5 User Guide for a detailed explanation of the HDF5 metadata cache behavior (http://www.hdfgroup.org/HDF5/doc/Advanced/MetadataCache/index.html).

At first glance, both of these implementations appear to guarantee the sequential consistency of access to the file's metadata, since there can be no conflicting access by different processes to the same metadata at once. However, considering the default consistency semantics of MPI-I/O (which is the interface used to read/write metadata on disk for parallel HDF5), sequential consistency can be broken with one particular access pattern:

1. All processes have a dirty metadata entry M in their cache.
2. Using either approach described above, metadata entry M is flushed to disk by process P and marked as clean on all processes.
3. Process Q evicts metadata entry M because it needs more space in the cache.
4. Process Q needs metadata entry M again, and since it was evicted it needs to read it from disk

Process Q was not the process that wrote metadata entry M to disk, so there is no guarantee that metadata entry M written out by process P actually hit the disk before the read call on process Q is issued.

To put this in an HDF5 example, consider the following scenario:

Process 0	Process 1
1) H5Acreate()	1) H5Acreate()
2) H5Awrite()	2) H5Awrite()
3) …	3) …

4)	4) H5Aread()

All processes create an attribute and write data to that attribute collectively. Process 1 comes in later and reads the same data that was written in the collective write call. Suppose that before process 1 issues the H5Aread() call, a metadata flush is triggered and using either approach A or B (described earlier), process 0 ends up writing the metadata of the attribute to disk, and then informs process 1 that this piece of metadata is clean, so process 1 evicts it from its cache. When process 1 calls H5Aread(), it goes back to disk to read the metadata for the attribute, specifically the data written in step 2. However, since MPI, in its default mode, is not atomic and does not guarantee sequential consistency, process 1 might actually read the old, the new one, or even an undefined value. This scenario can be translated to the following simple access to the same region in the file:

Process 0	Process 1
1) MPI_File_write_at()	1)
2) MPI_Barrier()	2) MPI_Barrier()
3)	3) MPI_File_read_at ()

In this case, as explained in the previous section, MPI-I/O does not guarantee that process 1 will actually read what process 0 has written, and therefore the consistency semantics for metadata operations are not guaranteed to be sequential with the current implementation.

One solution to the above issue is to enable MPI atomic mode. With atomic mode enabled, all file operations are guaranteed to execute with sequential consistency semantics. The major issue with atomic mode is that the performance drops significantly when reading/writing to the file.

The race condition described above is rare and has not yet been reported by any application that has used parallel HDF5. While it's important to handle the issue, it might be valuable to switch the metadata writes from the cache to be collective first, and then look at the consistency issue later, when more efficient solutions may be available in later releases of MPI. Additionally, this issue will disappear when the metadata server implementation we are currently working on is deployed.

PHDF5 Raw Data Consistency Semantics

As mentioned previously, the consistency semantics of data access is dictated by the virtual file driver used. In the parallel HDF5 library, when the MPI-I/O driver is used, the MPI-I/O consistency semantics model is enforced. HDF5 uses the default MPI semantics which does not guarantee sequential consistency. This means that access to a file happens in a non-atomic and arbitrary order. Although MPI-I/O provides an option for atomic (by enabling atomicity) and ordered access to the file (using sync-barrier-sync or atomicity-barrier), the parallel HDF5 library does not provide this functionality through its API.

A read after write on different processes (a process reading the data that another process wrote) can be achieved with the current API only by closing the file after the write and reopening it. Otherwise, a dataset read operation following a write (to the same dataset elements by a different process) is considered undefined.

Recommendation

The solution to provide stricter consistency semantics for file access is to provide the user with the same functionality that MPI offers to impose different levels of consistency. This would require adding the following two functions to the HDF5 API:

1) H5Fset_mpio_atomicity (hid_t file_id, hbool_t flag): enables (flag = 1) or disables (flag = 0) atomicity on a file opened with the MPI-IO VFD
2) H5Fsync (hid_t file_id): this call causes all previous writes to the file to be transferred to the storage device

Setting atomicity to true will call the MPI atomicity function (MPI_File_set_atomicity) which sets atomicity to true on the MPI file handle. The sync function will also call its MPI counterpart (MPI_File_sync). Using the two routines described above, the user will be able to enforce stricter consistency semantics, similar to what MPI provides. To be able to guarantee sequential consistency among different processes, the use of barriers (MPI_Barrier) will be required by either:

1) setting atomicity to true, then calling MPI_Barrier() to synchronize processes after writes/before reads

2) using the sync-barrier-sync construct

In certain scenarios, however, just setting atomicity to true in the MPI-I/O file driver does not suffice, because an H5Dwrite (for example) may translate into multiple MPI_File_write_at functions underneath. This happens in all cases where the high level file access routine translates to multiple lower level file access routines. The following scenarios will encounter this issue:

- Non-contiguous file access using independent I/O
- Partial collective I/O using chunked access
- Collective I/O using filters or when data conversion needs to happen

In any of these scenarios, the following case:

Process 0	Process 1
1) H5Fset_mpio_atomicity (fid, true)	1) H5Fset_mpio_atomicity (fid, true)
2) H5Dwrite()	2) H5Dread()

might not yield the expected result (the read happens as if the write was executed before or after it) if H5Fset_mpio_atomicity is implemented by just setting MPI-I/O atomicity to true, because it could translate into the following:

Process 0	Process 1

MPI_File_set_atomicity (fh, 1)	MPI_File_set_atomicity (fh, 1)
MPI_File_write_at() MPI_File_write_at() MPI_File_write_at() …	MPI_File_read_at() MPI_File_read_at() MPI_File_read_at() …

This means that atomicity is done per MPI file access operation and not per HDF5 operation, which is probably not what the user wants. The use of barriers after the H5Dwrite() and before the H5Dread() in addition to setting atomicity to true in the user's application would ensure that the H5Dread() and H5Dwrite() operations are atomic:

Process 0	Process 1
1) H5Fset_mpio_atomicity (fid, true)	1) H5Fset_mpio_atomicity (fid, true)
2) H5Dwrite()	2)
3) MPI_Barrier()	3) MPI_Barrier()
4)	4) H5Dread()

This does however also impose additional ordering semantics, where the operations are not only atomic, but also that the H5Dread() operation will occur after the H5Dwrite() operation, since the barrier will guarantee that all underlying write operations will execute atomically before the read operations starts.

Revision History

February 27, 2011: Version 1 circulated for comment within The HDF Group.