RFC: Enabling Collective Metadata Reads

Mohamad Chaarawi
Quincey Koziol

HDF5 uses a metadata cache internally for fast metadata access. Parallel HDF5 uses the metadata cache in the same way but with a requirement of having the same stream of dirty metadata from all MPI ranks. This means that all operations that modify metadata in the file are required to be collective. On the other hand, operations that read metadata from the file are treated as independent operations because they are allowed to be independent. On a cache miss, this will result in a small read from the file system.

It is quite common in HPC applications, although not always, that reading metadata is done collectively. For example, all processes opening groups or datasets to access them. This would result in small reads requests from all MPI ranks to the file system, reading the same data. This is obviously bad, since file systems perform horribly with such access patterns. This RFC proposes a new feature for users to tell the HDF5 library that all or certain metadata read operations are collective, which would allow the library to perform optimizations when reading the data on cache misses, by having one rank read the data and broadcasting it to all other ranks.

1 Introduction

The HDF5 metadata cache is a very important component of the library that enables fast access to file metadata instead of issuing multiple small accesses to the file system. Parallel HDF5 uses the same cache on every MPI process with some restrictions. The most important one being that all modifications to the file metadata have to be done collectively on all ranks. For more information about the metadata cache and requirements, consult the metadata cache user guide [1]. Writing metadata in HDF5 results from operations that modify the file structure, creating new objects, extending datasets, etc… Reading metadata results from operations such as opening the file and objects in that file, iterating through the file hierarchy, reading attributes, etc…

Reading HDF5 metadata does not modify the file and does not dirty any entries in the metadata cache. Thus, in Parallel HDF5, API operations that result in metadata reads are not required to be collective. Several applications take advantage of that fact for accessing different parts of the file on different MPI ranks. However, at the same time, many other HPC applications open the same objects or traverse the same file path to access data at a particular point. For example a very common use case in HPC applications is for all ranks to open a dataset at a particular location in the file and collectively read or write data to the dataset. This means that all ranks treat the API operation for opening a dataset collectively. At the moment, the HDF5 library treats all such operations as

independent operations, since they could very well be. This RFC proposes to add a new feature that lets users specify if a certain (or all) operations on the file are independent or collective.

2 Motivation & Approach

Applications that read a lot of metadata collectively will most probably see a big performance hit since the HDF5 library treats those reads as independent. This means that all ranks will independently read the same small data entries from the file system on a cache miss. These reads typically range from tens to hundreds of Bytes. Typical parallel file systems such as GPFS perform horribly with such an access pattern from all ranks.

The above scenario can be avoided with a very simple optimization by having one rank issue the read for a metadata entry to the file system and broadcast the data corresponding to the metadata cache entry to all other ranks. This would work if all the ranks are there to issue the corresponding broadcast operation. Adding a hint to the HDF5 library to indicate that an operation is executed collectively by the user would give the HDF5 library information that it can do such an optimization.

Applying the simple optimization we described above to the metadata cache is not as simple as it sounds, and a lot of infrastructure needs to be added to support this. More information about technical challenges of implementing this in the library is detailed in Section 3 for those who are familiar with the HDF5 metadata cache implementation.

2.1 New Property List Functions

The hint to the HDF5 library is passed from applications using a property on access property list. If the property to access metadata collectively is set to true on a file access property list that is used in creating or opening a file, then the HDF5 library will assume that all metadata read API operations issued on that file handle are going to be issued collectively from all ranks. Alternatively, users may wish to avoid setting that property globally on the file access property list, and individually set it on access property lists (dataset, group, link, datatype, attribute access property lists) for certain API operations. This will indicate that only the operation issued with the access property list is going to be called collectively.

The new routines for setting/getting the new property are:

herr_t H5Pset_all_coll_metadata_ops (hid_t plist_id, hbool_t is_collective);

herr_t H5Pget_ all_coll_metadata_ops (hid_t plist_id, hbool_t *is_collective);

Where plist_id is the access property list and is_collective is the value to turn on the optimization (set to 1) or turn it off (set to 0). The default is off, meaning operations are assumed to be independent.

2.2 Current HDF5 API Limitations

Some HDF5 API functions that read metadata from the cache and/or file system do not take any kind of access property list. This means that the property to indicate whether the library should do the metadata read optimization cannot be passed from the application through those functions individually but can be done on the file access property as a global hint for all functions. In future versions of HDF5, we should add a new version of those API routines adding an access property list.

The API routines that need a new version with an access property list added are:

H5Awrite()
H5Aread()
H5Arename()
H5Aiterate2()
H5Adelete()
H5Aexists()

H5Dget_space_status()
H5Dget_storage_size()
H5Dset_extent()
H5Ddebug()
H5Dclose()
H5Dget_get_create_plist()
H5Dget_space() (when the dataset is a virtual dataset)

H5Gget_create_plist()
H5Gget_info()
H5Gclose()

H5Literate()
H5Lvisit()

H5Rcreate()
H5Rdereference2() (If the referenace is an object reference)
H5Rget_region()
H5Rget_obj_type2()
H5Rget_name()

H5Ocopy()
H5Oopen_by_addr()
H5Oincr_refcount()
H5Odecr_refcount()
H5Oget_info()
H5Oset_comment()
H5Ovisit()

H5Fis_hdf5()
H5Fflush()
H5Fclose()
H5Fget_file_image()
H5Freopen()

H5Fget_freespace()
H5Fget_info2()
H5Fget_free_sections()
H5Fmount()
H5Funmount()

H5Iget_name()

H5Tget_create_plist()
H5Tclose()

H5Zunregister()
most deprecated routines

3 Implementation and Challenges

Note to readers: You must be familiar with the metadata cache V3 implementation to understand all the parts in this section.

Implementing the new optimization isn't a straightforward task. For example, checking if the property is set and simply doing a read from rank 0 and MPI_Bcast to other ranks won't work for the following reasons:

1) The caches on all the ranks contain the same list of dirty metadata entries. Clean metadata entries are not required to be the same on all ranks. The reason is that some metadata read operations could be done independently on some ranks. So consider for example in a collective read operation, rank 0 request access to cache entry X that is not in its cache but rank 1 has it in its cache as a clean entry. This means that rank 0 will read the entry from disk and bcast it to all ranks, but rank 1 won't enter that bcast because it sees that it has the entry in its cache, resulting in an application hang in the MPI_Bcast() operation.
2) Chunked dataset access, while it can be done independently or collectively, needs to access entries in the metadata cache independently for the chunks requested to be accessed by some processes. Since it is very unlikely that applications will issue collective access to the same pieces of raw data on 2 or more ranks, this means that even if we set the global hint on all operations on the file to be collective, there are some metadata read operations (chunk address lookups) that are required to be independent.

To address the above issues, some modifications were required to the internal cache implementation to support the collective metadata read optimization.

Each cache entry will be marked as accessed collectively or not. Once an item has been accessed collectively and marked so, it is added to a new LRU in the cache that holds previously collectively accessed entries. The collective LRU has the following two requirements that have to be maintained at any given point in time:

1) The number of entries on all ranks should be the same
2) The order of all entries in the LRU list should be the same on all ranks.

An entry marked collective can't be unmarked or evicted from the cache independently by any rank. Furthermore, if a rank independently accesses an entry in the collective LRU list, its order is not changed in the LRU list. Only if the access is collective, the entry is moved to the top of the LRU.

On a cache sync point, where all processes collectively get together to clean entries in the cache and possibly evict some, all ranks will unmark the bottom half of the collective entries in the collective LRU list an remove them from the list. Note that this does not evict those entries from the cache, but will allow them to be evicted by the normal cache replacement policy. The sync point however is not triggered in read only scenarios, where there are not dirty entries in the cache. In this case, we check on every protect operation on entry if it is done collectively. If yes, then we compare the total byte size of the collective entries in the collective LRU to the maximum cache size. If the collective entries grow up to a certain threshold point from the maximum cache size, we clear the bottom half of the collective LRU and mark those entries as independent and allow them to be flushed. The threshold now depends on the global collective metadata read setting on the entire file:

- If the application indicates that all operations on the file metadata are collective, we allow the collective metadata entries to grow to 80% of the cache size.
- Otherwise we allow them to grow to 40% of the cache size.

With the above changes in place, we can now proceed to implement the optimization itself. On a protect call to read a metadata entry, all ranks check if the read is done collectively. If it is not, the original behavior does not change. If yes, each process checks whether the entry is in its cache. The following scenarios are possible:

- The entry is in the cache and is marked as collective. In that case, no more work is needed since all processes will have it in the cache since it is marked as collective.
- The entry is in the cache, it is clean, but is not marked as collective. This means that other ranks might not have the entry in the cache. The broadcast operation is issued on all ranks including the ones that have the entry in their cache. Process with rank 0 is responsible for reading the entry from disk if it is not in its cache and broadcasting the entry to all the ranks.
- The entry is in the cache and is dirty. This means that all processes will have in its cache. If it is not marked as collective, we mark it as so and we insert it in the collective LRU.

4 Recommendation

Reading the same data by all ranks is definitely a bad approach when it can be avoided since the network is a much faster means for sharing the data from 1 rank. This RFC aims to address that limitation in HDF5 by allowing application users to pass a hint to the HDF5 library that all or specific read operations on the file are collective. The HDF5 library would have 1 rank read the data and broadcast it to all ranks to avoid burdening the file system.

The changes to support this feature were implemented on the metadata cache merge branch where the new Version 3 of the metadata cache has been developed.

Revision History

May 15, 2015:	Version 1 circulated for comment within The HDF Group.
February 10, 2016:	Version 2 describes updated read operation functions.

[References]

[1] Metadata Caching in HDF5: http://hdfgroup.org/HDF5/doc/Advanced/MetadataCache/index.html