RFC: Page Buffering

Mohamad Chaarawi

The HDF5 library generates metadata I/O along with application raw data I/O when accessing a file. The total amount of metadata in an HDF5 file varies widely between applications, but one common aspect across most applications is that metadata accesses are usually small. Additionally, some applications access raw data in HDF5 datasets in very small amounts.

Small and random I/O accesses on parallel file systems result in poor performance for applications. HDF5 metadata and tiny raw data accesses fit this description. There has been a significant amount of work done to avoid such small accesses in HDF5, but more opportunities exist for improvement in this area. This RFC proposes a new feature called Page Buffering. Page buffering in conjunction with Paged Aggregation (a feature already implemented in HDF5) will ensure that HDF5 I/O requests to a file can be minimized to a specific granularity and alignment, controlled directly by the application.

1 Introduction

HDF5 is used by many applications on a variety of platforms. Science applications on HPC systems utilize HDF5 for reliable and efficient data access on parallel file systems. However, HDF5's rich data model and hierarchical nature can generate a moderate amount of metadata to store in the HDF5 file, along with raw data from the application. The total amount of metadata varies widely between applications depending on their usage of the HDF5 data model, but one commonality is that metadata accesses are usually tiny. HDF5 thus employs a metadata cache to try and avoid swarms of small accesses to the file system, but at some point metadata needs to be evicted or flushed from the cache, resulting in small accesses to the file system.

Other applications access raw data in small amounts. The HDF5 datasets created by those applications might be small, or the selections for partial I/O on large datasets might be small. Either way, small accesses to the file result from such access patterns. HDF5 provides a cache feature for raw data, but that is available only for chunked datasets and is disabled in parallel HDF5. HDF5 can also do data sieving for contiguous datasets, but that is disabled in parallel HDF5 too.

Parallel file systems on HPC machines, such as Lustre and GPFS, perform very poorly with small, fragmented I/O accesses. Recent work in the Parallel HDF5 library attempts to circumvent that problem by writing HDF5 metadata collectively and reducing the number of metadata reads by allowing application to indicate that reads to certain pieces of metadata will be done collectively which enables an optimization to read metadata only from 1 rank. Another feature implemented to improve small file I/O performance was to enhance the file space allocation in the HDF5 library to make it page aligned, i.e. allocating space in fixed sized pages and aggregating as much metadata or raw data as possible in a single block before allocating another block. This prevents metadata from possibly being scattered in small pieces all around the file.

This work will leverage the paged aggregation of file space allocation to develop a page buffering layer in the HDF5 library that absorbs small accesses to the file system. Each page in memory corresponds to a page allocated in the file. Access to the file system is then performed as a single page or multiple of pages, if they are contiguous. This ensures that small accesses to the file system are avoided while providing another caching layer for improved I/O performance (albeit at the expense of higher memory usage).

2 Approach

Every I/O access to the file system generated by HDF5 can be classified as a metadata or raw data operation. For metadata operations, the HDF5 metadata cache works to reduce the number of small accesses to the file system. However at some point in time, whether at a user requested or internally triggered flush of the metadata cache, entries in the cache are written out to disk. In addition, depending on the application's access pattern, new metadata is periodically brought into the cache by reading it in from disk. For raw data, access is done directly to the file system, unless the chunk cache is enabled (in configurations where it is supported).

The page buffering feature proposed in this RFC will add a layer immediately above the Virtual File Driver I/O layer within the library, and act as a page cache for metadata and raw data accesses that are smaller than the page size for file space allocation. I/O from the page buffer is always done in multiples of pages. For example, if the page size is set to 16 KB, no I/O to the file system from HDF5 will be smaller than 16 KB.[1]

The page buffering feature is built on the paged aggregation feature that was previously prototyped by the HDF5 group. Paged aggregation provides users with option of indicating to the library when creating an HDF5 file that file space allocation method used by the library should be done in fixed sized pages or blocks. Each block will contain either metadata or raw data (but never both). If the page size is large enough, metadata (for example) will be aggregated as much as possible into a single block before allocating another block for storing more metadata. This prevents having metadata scattered around the file, which might cause a lot of random I/Os when accessing metadata. However, this still does not change the actual size of the I/O done to disk, so small accesses are still sent to the file system.

Building on the paged aggregation feature, page buffering allows users to specify a maximum page buffer size to store paged size buffers in memory. The total size of all buffered pages should be set to an exact multiple of the paged aggregation size, and each buffered page's size is equal to the file's page aggregation size. Then, when HDF5 requests to read a small (smaller than a page size) piece of data from the VFD layer, the page buffering layer will check if the page that holds that piece exists in the page buffer. If so, it can satisfy the read from that page in memory; otherwise the page buffer reads the entire page containing that piece and caches it. Writing small data pieces similarly checks if the page that contains that piece is already in the page buffer. If it is, then the page buffer updates

[1] Except a few operations that we discuss further in the technical section, and if the total metadata or raw data does not exceed 16 KB

the page; otherwise it reads in the page from disk, updates it, and caches it for future use. The read can be avoided if we are writing to a newly allocated page that we can track from the file space manager. Reads and writes of large (larger than a page size) metadata will bypass the page buffering layer completely since those accesses are atomic and large enough to not require paged access. For raw data, extra processing at the page buffer layer is required, since raw data accesses are not as atomic as metadata accesses. This is discussed further in the technical section below.

3 Use cases

It is well proven that parallel file systems on HPC systems perform badly with small fragmented I/O accesses. Considering HDF5's metadata access is usually small in the overwhelmingly majority of cases, almost all HDF5 applications will benefit from page buffering. Some applications will benefit more than others, depending on the amount of metadata they generate through their usage of HDF5. Our target for this feature is mainly applications using serial HDF5 on HPC systems.

Parallel applications that use serial HDF5 in Multiple Independent Files (MIF) I/O method should see a great improvement, because the number of small accesses will be reduced significantly. Examples of MIF include the "one file per process" I/O method and the "poor man's parallel I/O" method used at LLNL.

The Single Shared File (SSF) I/O paradigm (aka "Rich man's parallel I/O"), or in other words parallel HDF5 using MPI-IO VFD, will also benefit from this feature, but as we will see later there are some limitations to what can be supported with parallel HDF5 and Page Buffering. Furthermore, parallel HDF5 has other new features that attempt to overcome the limitations of small metadata accesses.

4 API Additions

To use page buffering, the file space allocation strategy must first be set to H5F_FSPACE_STRATEGY_PAGE, using this API routine:

```
herr_t H5Pset_file_space_strategy(hid_t fcpl_id, H5F_fspace_strategy_t strategy,
hbool_t persist, hsize_t threshold);
```

- `fcpl_id: The file creation property list used to create a new file.`
- `strategy: An enum of the file space allocation strategy to be used. This can be set to:`
  - o `H5F_FSPACE_STRATEGY_PAGE: use free-space managers with embedded paged aggregation and virtual file drivers.`
  - o `H5F_FSPACE_STRATEGY_FSM_AGGR: use free-space managers, aggregators and virtual file drivers (library default).`
  - o `H5F_FSPACE_STRATEGY_AGGR: use aggregators and virtual file drivers.`
  - o `H5F_FSPACE_STRATEGY_NONE: just use the virtual file drivers.`
- `persist: A bool to indicate whether free space should be persisted or not.`
- `threshold: A value the free space manager(s) will use to track free space sections.`

Then the size of the file space allocation page (and the I/O size that will result from the page buffer if page buffering is enabled) can be specified using:

```
herr_t H5Pset_file_space_page_size(hid_t fcpl_id, hsize_t fsp_size);
```

- `fcpl_id: The file creation property list used to create a new file.`
- `fsp_size: The file space page size when file space allocation strategy is H5F_FSPACE_STRATEGY_PAGE.  The default size is 4096.`

The creation property list with the above properties set should then be passed to the file creation (H5Fcreate) routine. The following routines similarly retrieve the values from a file creation property list pertaining to file space allocation:

```
herr_t H5Pget_file_space_page_size(hid_t plist_id, hsize_t *fsp_size);
```

```
herr_t H5Pget_file_space_strategy(hid_t plist_id, H5F_fspace_strategy_t *strategy,
hbool_t *persist, hsize_t *threshold);
```

To enable page buffering, the user should then set the maximum page buffer size property on the file access property list using:

```
herr_t   H5Pset_page_buffer_size(hid_t   fapl_id,   size_t   buf_size,   unsigned
min_meta_per, unsigned min_raw_per);
```

- `fapl_id: the file access property list.`
- `buf_size: maximum page buffer size.`
- `min_meta_per: Minimum metadata percentage to keep in the page buffer before allowing pages containing metadata to be evicted (Default is 0).`
- `min_raw_per: Minimum raw data percentage to keep in the page buffer before allowing pages containing metadata to be evicted (Default is 0).`

The file access property list should then be used when creating or opening the file. Note that if page buffering is set in the FAPL, but paged allocation is not the allocation strategy in the FCPL, the file's create or open call will fail. Setting the maximum page buffer size smaller than the paged allocation size will also cause the create/open call to fail. The maximum page buffer size should be a multiple of the paged allocation size, but if it's not an exact multiple, the library will round down to the next lower exact multiple. However if the page buffer size is set to a value lower than that of the paged allocation size, the file create or open routine will fail, since it won't be the user's intention to set a page buffer and have the library round it down to zero, meaning there will be no page buffering done at all.

The corresponding "get" routine retrieves the page buffer size from the file access property list:

```
herr_t   H5Pget_page_buffer_size(hid_t   fapl_id,   size_t   *buf_size,   unsigned
*min_meta_per, unsigned *min_raw_per);
```

5 Implementation and Technical Details

The page buffer will be a new layer within the HDF5 library that sits just above the VFD layer. This will capture all I/O calls that are issued in the library, with information on:

- The type of the operations (read or write)

- The type of HDF5 data being accessed (raw data or metadata)
- The address of the I/O request in the file
- The size of the I/O request
- The buffer for the data to be written from or read into.

This new layer (with routines prefixed by 'H5PB') will allow creating a page buffer cache for the file. The page buffer for a file contains information about maximum buffer size, the skip list containing all the buffered pages, the LRU list for tracking which pages to evict, etc.

When creating a new file, assuming paged allocation strategy and page buffering are both enabled, the page buffer cache will initially be created with no pages. Similarly, if page buffering is requested when opening an existing file, a new page buffer cache is instantiated with no pages; however before doing that, the file's superblock must be read to retrieve configuration information about the file, with the paged allocation size being one of them. This is the only I/O operation that will occur outside of the page buffering scheme, since the page buffer can't be set up before the file superblock is retrieved from an existing HDF5 file.

5.1 Page Buffer Operation

When an I/O is generated, whether through an application raw data access or a metadata access requested from the metadata cache, the request goes through the page buffer layer. If the I/O request is equal or larger than the file's page size, the page buffer will call directly into the VFD layer[2]. If the access size is smaller than a page, then the page buffer takes on the responsibility for satisfying that request.

5.1.1 Metadata

On a metadata read the following happens in the page buffer:

1. Calculate the address of the page that will contain the request, using the address of the I/O request and the file's page size.
2. Search the skip list of pages to determine if the page required is in the PB cache.
3. If it is, copy the data from the page into the request's buffer, move the page to the top of the LRU, then return.
4. If the page is not in the PB cache, determine if we have space for a new page. If we do, skip to step 6.
5. If there is no space in the PB cache for another page, evict one (or more) "cold" pages from the bottom of the LRU to make space, by writing them to the file if they are dirty, or just discarding them if they are clean. A dirty page can also be discarded instead of written if it falls beyond the EOA (End of Allocation) address for the file since it means that this page of data has been deleted from the file. Note that before just evicting the coldest entry in the LRU, we check the minimum metadata and raw data settings to make sure we keep within the minimum limits that the user sets.

[2] There are additional steps required for raw data that we detail later in this section.

6. Read the new page from disk, insert it into the skip list, and to the top of the LRU list.
7. Copy the data from the new page into the request's buffer and return.

On a write for metadata the following happens:

1. Calculate the address of the page that will contain the request, using the address of the I/O request and the file's page size.
2. Search the skip list of pages to determine if the page required is in the PB cache.
3. If it is, copy the data from the request's buffer into the page, move the page to the top of the LRU, then return.
4. If the page is not in the PB cache, determine if we have space for a new page. If we do, skip to step 6.
5. If there is no space in the PB cache for another page, evict one (or more) "cold" pages from the bottom of the LRU to make space, by writing them to the file if they are dirty, or just discarding them if they are clean. A dirty page can be also discarded instead of written if it falls beyond the EOA (End of Allocation) address for the file since it means that this page of data has been deleted from the file. Note that before just evicting the coldest entry in the LRU, we check the minimum metadata and raw data settings to make sure we keep within the minimum limits that the user sets.
6. Determine whether the page is newly allocated through the hint from the file space manager. If it is, there is no need to read the page from the file. Just create a new page in memory with a newly allocated and zeroed buffer. If it is not, read the page from disk.
7. Insert the page it into the skip list, and to the top of the LRU list.
8. Copy the data from the request's buffer into the new page and return

On freeing a page of metadata, which is resulted from merging small-sized (smaller than a page) metadata sections that are freed, the following happens in the page buffer:

1. Search the skip list of pages to determine if the page is in the PB cache.
2. If the page exists in the PB cache, remove the entry from the skip list and the LRU list.

5.1.2 Raw Data

Reads and writes for raw data will be similar to metadata, but require additional steps. Metadata accesses in HDF5 files are "atomic", which means that accessing a metadata entry is always performed on the entire entry. Raw data is different and can be accessed in many different portions as the user requests with HDF5 selections.

When writing raw data, the library must make certain to check if the write request spans pages that are already in the PB cache if the I/O size is large enough to bypass the PB, after we call the VFD write. The page buffer needs to update the pages in the PB that are partially accessed by the large I/O write and discard the ones that are completely overwritten; otherwise subsequent reads will read invalid data from the page buffer.

If the size of the raw data read is larger than the page size, we issue the VFD read to get the entire block requested and then check if the I/O request spans pages in the PB cache. If it does, and those pages are dirty, then we need to update the portion of the read buffer with the dirty data from the PB; otherwise the read would be returning outdated data.

5.2 Parallel HDF5 Considerations

Parallel HDF5 (SSF method) has more issues to take into consideration, and based on these issues we will limit the support for page buffering. The most important thing to keep in mind for parallel HDF5 is that all processes are required to see a consistent view of metadata and raw data as if the I/O has been done to the file system.

5.2.1 Metadata

All ranks will see the same stream of dirty metadata in the metadata cache due to the fact that all HDF5 operations that modify metadata in the file are required to be collective. On a flush of the metadata cache, the page buffer can be utilized when the flush strategy is set to H5AC_METADATA_WRITE_STRATEGY__PROCESS_0_ONLY (Rank 0 does the metadata writes). In that case the following will need to happen:

1. “Cork” the page buffer cache on rank 0 (so that pages are not evicted)
2. Rank 0 writes all the dirty entries (into the page buffer).
3. Rank 0 tracks all the pages that are being touched by the metadata flush.
4. Once rank 0 writes all the metadata entries, it “uncorks” the page buffer cache and flushes all the pages that have been touched.
5. The other ranks then update pages in their page buffer with the dirty metadata entries that Rank 0 wrote. (Pages that rank 0 wrote that aren’t in their buffer will be re-read from the file if needed in the future)

This ensures that future reads of metadata from all ranks is consistent with what has been flushed into their PB and to the file.

In the distributed metadata write strategy, where the dirty metadata cache entries are distributed between all the ranks to write, it is possible to utilize the page buffer, however it requires a new distributed algorithm to distribute entries according to pages (no two ranks should have entries that touch the same page). Furthermore all the touched pages must be flushed and the existing ones updated with what other ranks have written. We think that the overhead of communication and actual complexity to do that might not be worth the effort.

On the other hand, a new optimization will be added soon to the HDF5 library that allows a flush to write all the dirty metadata entries collectively with `MPI_File_write_at_all()`. Preliminary results show that this enhancement yields huge performance benefits over the existing approach. We expect that this would be better than doing any page buffering for metadata writes and MPI would handle this for us by doing collective aggregation between the ranks using two-phase I/O or other collective I/O strategies to aggregate small data into larger I/O requests.

Another feature that will soon be added to HDF5 is a capability to allow users to indicate whether a metadata read operation is issued collectively by the processes. If that is the case, then the library will have process 0 perform the reads and broadcast the metadata entries it reads to the other ranks. Adding a PB layer in that case should enhance the performance of this feature even more by reducing the overall number of times that process 0 reads from the file (with many reads actually being retrieved from a buffered page). To allow metadata reads to use the page buffer, the metadata cache has to update existing pages after a metadata sync point that flushes dirty entries to the disk on all ranks accessing the file. This is easily doable since at the sync point, all processes already have the list of need to be updated entries, even though the actual metadata updates are done by process 0 or distributed among all participating processes.

5.2.2 Raw Data

Accessing raw data with parallel HDF5 creates many problems for the page buffer layer because on top of the fact that raw data access is not atomic like metadata, raw data accesses can be performed collectively or independently by the application.

When accessing raw data collectively, the HDF5 library constructs MPI derived datatypes to represent possibly non-contiguous buffers in memory and file offsets. At the page buffer layer, we would be required to flatten the constructed derived datatypes, which poses a considerable overhead, update all the page buffers, and then impose additional communication among all processes to determine who reads/writes what pages for the collective operation. We think MPI already does a good job at these operations, and bypassing the page buffer would be much better than doing MPI’s job inside HDF5. Furthermore applications accessing raw data collectively usually have enough data among all the ranks for an access operation to not be small enough to cause problems on parallel file systems.

Independent raw data access with parallel HDF5 could benefit from page buffering, however since both collective and independent operations can be used at the same time in applications, there would be data inconsistencies with the page buffer between the collective and independent operations. So we have decided to bypass the page buffer for independent raw data accesses too.

5.3 Single Write Multiple Reader (SWMR)

SWMR allows multiple reader applications to read the HDF5 file while a write application is updating it. One of the requirements here is that when the write process writes to the HDF5 file in a certain order, the writes have to happen to file in the exact order they have been called. This means that we cannot hold those writes in the PB because we will lose the required ordering aspect of the write. So for the writing app, page buffer is just a write through to the VFD.

Readers can operate normally (using the page buffer), until a “refresh” operation is performed on an object. When a refresh occurs, all the entries evicted from the metadata cache must also have their page evicted from the page buffer. Then, the following read operations will bring in the correct data from the writer.

6 Statistics Gathering

The page buffer layer gathers some statistics about page access when it is enabled, similarly to what the metadata cache does. The user can retrieve those statistics with the following new API routine:

```
herr_t H5Fget_page_buffering_stats(hid_t file_id, int accesses[2], int hits[2],
int misses[2], int evictions[2], int bypasses[2]);
```

- `file_id`: file identifier.
- `accesses[2]`: 2 integer array for the number of metadata and raw data accesses to the page buffer.
- `hits[2]`: 2 integer array for the number of metadata and raw data hits in the page buffer.
- `misses[2]`: 2 integer array for the number of metadata and raw data misses in the page buffer.
- `evictions[2]`: 2 integer array for the number of metadata and raw data evictions from the page buffer.
- `bypasses[2]`: 2 integer array for the number of metadata and raw data accesses that bypass the page buffer.

The user can also reset the page buffer stats using:

```
herr_t H5Freset_page_buffering_stats(hid_t file_id);
```

7 Regression Testing

This section describes how we do regression testing on the page buffering feature. Note that we do not describe in detail what each test does, but only provide a general idea of what is tested. For more details, look at the page buffering test code.

7.1 Serial Regression Tests

A new test file (tpage_buffer.c) is added to the HDF5 serial test suite containing all the page buffer serial regression tests. The test is divided into the following subcategories:

1. Argument testing (test_args): This tests arguments validity to setting page buffer usage. For example, the test makes sure that files create operation fails when setting page buffering if paged aggregation is not enabled. Furthermore corner cases are testing in case of very small page buffers, page buffer size equal or slightly greater than the paged aggregation size, etc… The valid arguments are testing with 2 functions, one to create an HDF5 file with access to a group and datasets, and the second to open the created file from the previous function and access it again.
2. Handling the Raw Data updates in the page buffer (test_raw_data_handling): As we mentioned in section 5.1.2, raw data writes through the page buffer needs to update or invalidate existing pages even if it is a bypass, and raw data reads need to get updates from the page buffer on a bypass. This test checks correctness of the page buffer code in doing those updates.
3. LRU processing correctness (test_lru_processing): This test verifies the correct ordering of page buffer entries in the LRU list that governs the eviction policy. It verifies that we never exceed the maximum number of pages in the page buffer and the coldest pages are always evicted first.
4. Minimum metadata and raw data threshold testing (test_min_threshold): This test verifies that the minimum metadata and raw data settings are always maintained in the page buffer and the LRU eviction scheme takes into account those settings before evicting any entry.

5. Statistics Collection testing (test_stats_collection): This verifies accurate statistics gathered in the page buffer with a specific access to the page buffer.

7.2 Parallel Regression Tests

We update the file tests (t_file.c) in the testpar directory for parallel regression tests to verify the correctness of page buffering usage in parallel HDF5 (SSF with the mpio file driver).

First we verify that access to an HDF5 file (similar test scenario to the serial test cases #1) works correctly in parallel when page buffering is enabled with both metadata write strategies (Process 0 and Distributed). To test correctness of metadata access as described in 5.2.1, the second test to open an existing file and access it is updated with a parallel specific scenario:

- Create a new group after the file is opened.
- Flush the file forcing all metadata entries to be written out (from process 0 or distributed) to disk. The metadata writes will update the page buffer entries only from those processes that are responsible for writing those entries. The dirty pages in the page buffer are also written out.
- Remove all metadata entries from the metadata cache since everything is clean now.
- Try to open the created group earlier.

If all processes do not update their pages when the metadata entries are written out in the flush operation, the group open will fail on those processes that did not do the page buffer updates, since they will try to read out of date metadata from the page buffer.

The parallel test also verifies that an application running with 1 MPI rank leverages the page buffer like any serial application does, since there are no coordination issues. Raw data access does not bypass the page buffer in that case.

Finally we verify that application with more than 1 process do bypass the page buffer on raw data and metadata writes, but use the page buffer with raw data reads.

8 h5repack: Change Page Size

New options are added to *h5repack* to allow users to copy an existing HDF5 file (possibly created with no paged aggregation) to another HDF5 file with paged aggregation enabled with a user specific page size. The *h5repack* options added are:

- `-S FS_STRATEGY, --fs_strategy=FS_STRATEGY`
    - File space management strategy set via *H5Pset_file_space_strategy*
- `-P FS_PERSIST, --fs_persist=FS_PERSIST`
    - Persist or not persist free-space set via *H5Pset_file_space_strategy*
- `-T FS_THRESHOLD, --fs_threshold=FS_THRESHOLD`
    - Free-space section threshold set via *H5Pset_file_space_strategy*
- `-G FS_PAGESIZE, --fs_pagesize=FS_PAGESIZE`

- File space page size set via *H5Pset_file_space_page_size*

9 Recommendation

This RFC proposes to add a new layer in HDF5 above the VFD to capture small I/Os to the HDF5 file and hold them in paged size blocks in memory. Page Buffering would issue only page-sized and page-aligned I/O to the file system to avoid the costly performance of accessing small fragmented data on the file system. There will be I/Os with sizes larger than a single page that bypass the page buffer and will not be page aligned. The page size and page buffer size are tunable knobs that are set by the user. We discussed the benefits expected with page buffering in serial HDF5 usage in the MIF parallel I/O model and some of the limitations it has in the SSF parallel I/O model and SWMR scenario. The recommendation is to focus this feature for serial HDF5 and do what seems sensible with the required effort for parallel HDF5 and SWMR.

This feature will be implemented by the end of September 2015 and is expected to be released with the HDF5 1.10.0 release in 2016.

Acknowledgements

This work is sponsored by Lawrence Livermore National Laboratories. We thank Mark Miller for his support.

Revision History

June 25, 2015:	Version 1 circulated for comment to Quincey.
June 26, 2015:	Version 2 updated with comments from Quincey
July 09, 2015	Version 3 updated with comments from Mark Miller.
July 27, 2015	Version 4 updated with new API, regressions testing, statistics gathering, h5repack
March 6, 2017	Version 5 updated to reflect coding changes.