A "Private" Heap for HDF5

Quincey Koziol

Jan. 15, 2007

Background

The HDF5 library currently stores variable-sized data in two different data structures in its files. Variable-sized metadata (currently only the names of links) is stored in a "local" heap for each group (http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.format.html#LocalHeap). Variable-sized raw data for datasets (like variable-length and dataset region reference datatypes) is stored in the file's "global" heap (http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.format.html#GlobalHeap).

Limitations of Current Heap Implementations

Local heaps have several limitations that need to be overcome: the heap data is stored in a single block and the heap IDs used to locate entries in a local heap is the offset of the entry within the data block. Keeping the heap data in a single block requires that the data block be relocated in the file when it expands and currently requires that the entire data block be read from disk when any entry in it is accessed. Using the offset of the entry within the data block as the heap ID allows for fast lookups of the entry's data, but means that entries cannot be relocated within the data block.

Global heaps have different limitations to overcome: the heap's data blocks are shared among all global heap users (currently only the datasets) in the file, the data blocks are not tracked by any data structure and an entry in a data block is located with a linear search. Sharing the data blocks among all datasets in the file means that when a data block is brought into the cache in order to access a particular entry, it may contain only a single entry for the dataset being accessed. Additionally, sharing data blocks among all datasets in the file makes deleting the heap entries used by a dataset being deleted very difficult and time consuming (each element of the dataset being deleted must be accessed and the individual heap entry must be released). The lack of a data structure tracking all data blocks for a file's global heap means that there is no way to locate existing data blocks when none are cached and new entries need to be stored in the heap (requiring a new heap data block to be allocated). Locating an entry in a data block with a linear lookup is time-consuming when a data block is large and stores many small entries.

Motivation for a Redesigned Heap

As the format for groups in HDF5 is being redesigned currently (in preparation for the 1.8.0 release), it's a good time to update the design of the local heaps. Moreover, it would be nice if there were only one type of heap in the file, so the redesigned heap should attempt to supplant the global heaps as well as the local heaps. Each object in the file (dataset or group) that needed to store variable-length data would have it's own heap, hence the new heaps are called "private" heaps throughout the rest of this document.

Use Cases for a Private Heap

Several use cases for the new heap data structure arise from replacing the local and global heaps in the file:

- The new private heap should replace using a local heap for groups. Thus, it needs to fill the current group operation use cases, in rough order of priority:
    - Looking up an entry by it's heap ID should be very fast
    - Heaps should store small entries efficiently (most links for a group are estimated to be ~64 bytes in size)
    - Heaps should be able to store entries up to at least 64KB in size
    - The heap should have very small metadata overhead per entry (since the size of entries will be small)
    - Adding entries to the "end" of heap should be fast
    - The heap should allow storing 100,000+ entries
    - Deleting entries from the heap should be efficient
- The new private heap should replace using global heaps for storing variable-length data and region references in datasets. Thus, it needs to fill the current dataset operation use cases, in rough order of priority:
    - Adding entries to the "end" of a heap should be fast
    - Looking up an entry by it's heap ID should be fast
    - Heaps should store both large & small entries efficiently
    - Heaps should not impose upper limits on an entry's size
    - Heaps should not impose upper limits on the number of entries stored
    - Entries in the heap should be able to be reference counted (so fill values in datasets can be implemented efficiently)
    - Deleting entries from the heap should be possible
    - Deleting an entire heap should be possible without iterating over each element in a dataset
- Additionally, the existing local & global heaps are missing several capabilities that would be valuable in future development:
    - It should be possible to compress the entries in the heap
    - It should be possible to request storage of a collection of related entries, in a heap of a given size, to avoid fragmentation and improve I/O performance.

Design Goals for a Private Heap

To replace both the local and global heaps, the new "private" heap should achieve the following goals:

- The heap should be self-contained (not global to the entire file or required to be shared with other file objects)
- The heap should track all of it's data blocks (so that all the space in the file used by the object it belongs to can be accurately determined)
- IDs for entries in the heap should be persistent (adding or removing entries in the heap should not change the ID of other entries)
- Existing entries in the heap should be very fast to locate and read

- Allocating space for and storing new entries in the heap should be very fast
- The size of a entry should not be limited
- The number of entries in a heap should not be limited
- Entries should be able to be removed from the heap, but removal speed should not be a top priority
- Each entry should have an optional reference count

Design of the Private Heap

The new heap will be based on a novel data structure, the "doubling table", which is described in Appendix 1. An example of a doubling table with a width of 4 and a starting block size of 4096 is shown here:

Row	Blocks Added				Array Size
0	4,096	4,096	4,096	4,096	16,384
1	4,096	4,096	4,096	4,096	32,768
2	8,192	8,192	8,192	8,192	65,536
3	16,384	16,384	16,384	16,384	131,072
4	32,768	32,768	32,768	32,768	262,144
5	65,536	65,536	65,536	65,536	524,288
…					

IDs of objects in the heap are the linear offset of the entry in the heap, as if it were an array. So, an ID of 22384 would be located in the 2nd block in row 1, at offset 1904.

Each block in the doubling table represents a block in the heap, which the HDF5 library brings in from disk when it is accessed.

TODO:

- describe recursive/fractal nature of heap as larger blocks are used, especially as difference from "standard" doubling table, along with the resulting indirect/direct blocks.
- describe "tiny" object storage in heap ID itself
- describe "huge" object storage outside the heap
- describe optional direct block compression (huge objects too)
- followup with use case & design goals met or compromised on

Appendix 1 – Doubling Tables

A doubling table provides a mechanism for quickly extending an array data structure in a way that minimizes the number of empty elements in the array, while retaining very fast lookup of any element within the array. They are described in this appendix.

The classic way to increase the size of an array in memory is to double it's size each time the array needs to be extended. This approaches O(n) memory element copy operations and the usage of the array varies between 50% and 100% full. For example, see the table below: (QAK: could use picture)

Current Size	New Size	# of Elements Copied Now	Total # of Elements Copied	% Full Before Extend Occurs	% Full After Extend Occurs
256	512	256	256	100%	50%
512	1024	512	768	100%	50%
1024	2048	1024	1792	100%	50%
2048	4096	2048	3840	100%	50%
…					

This scheme may be reasonable for memory arrays, but is difficult for arrays on disk, due to the time involved for repeated data copying, especially as the size of the array increases. Additionally, because each new copy of the array is twice as large as the previous one, the existing space in the file can't be re-used, if it is not at the end of the file, leading to a file that could be twice as large as necessary.

However, if we think about an array more abstractly, as an ordered collection of homogenous elements, we could perhaps store those abstract elements in a set of blocks of elements, to facilitate more interesting array algorithms. Then, instead of copying the array elements each time the array's size is increased, it would be possible to leave the old array block where it was when a new block was added to the array and to build up an index for locating array elements in blocks that are added. For example, see the tables below: (QAK: could use a picture, showing index & blocks)

Block #	Block Size	Array Size	% Full After Extend Occurs
0	256	256	50%
1	256	512	50%
2	512	1024	50%
3	1024	2048	50%
4	2048	4096	50%

Element #'s	Block #
0 – 255	0
256 – 511	1
512 – 1023	2

1024 – 2047	3
2048 – 4095	4

This approach reduces the amount of element copying, but when the array expands, the new "array" is still 50% empty.

Now, instead of doubling the block size each time, if we keep the same block size as several blocks are added to the array, we could increase the percent of the array that is full when each block is added. For example, if we start with 256 byte blocks and have four blocks of the same size before doubling the size of the block to 512 bytes, etc., the blocks added and the percent full of the array would look like this: (QAK: could use a picture, showing relative block sizes, as the are added)

Block size	% Full when extend occurs
256	0
256	50%
256	66.6%
256	75%
512	66.6%
512	75%
512	80%
512	83.3%
1024	75%

(QAK: add another example, with a different width)

If this trend is continued, it can be seen that the percent full when a new size of blocks is added approaches 80% full when adding four blocks of the same size (a "width" of 4 – the "width" is the # of blocks added of the same size, before doubling to the next size). After some experimentation, the following table of percent full information was derived:

Doubling Table Width	Low % Full When New Block Added*	High % Full When New Block Added*	Low to High Range
1	50.00000%	50.00000%	0.00000%
2	66.66667%	75.00000%	8.33333%

4	80.00000%	87.50000%	7.50000%
8	88.88889%	93.75000%	4.86111%
16	94.11765%	96.87500%	2.75735%
32	96.96970%	98.43750%	1.46780%
64	98.46154%	99.21875%	0.75721%
128	99.22481%	99.60938%	0.38457%
256	99.61089%	99.80469%	0.19379%
512	99.80507%	99.90234%	0.09728%
1024	99.90244%	99.95117%	0.04873%

* - "In the limit" as 'infinite' array size is reached.

(QAK: show graph of % Full for several widths, as blocks are added)

Clearly, making the table "wider" benefits the percent full when adding new blocks to the array. Probably a point of diminishing returns is reached in the 16-64 range of widths. Also, this progression is only shown for widths that are powers of two. Other widths follow a similar progression, but are a poor match for implementing in software.

Additionally, in order to work out an easy algorithm for computing which block an array offset corresponds to, we found that it was better to make the first two "rows" of blocks in the table be the same size, before starting to double the blocks' sizes. An example for a table with width 4 is:

Row	Blocks Added				Array Size
0	256	256	256	256	1024
1	256	256	256	256	2048
2	512	512	512	512	4096
3	1024	1024	1024	1024	8192
4	2048	2048	2048	2048	16384
5	4096	4096	4096	4096	32768
…					

This corresponds to "rolling up" all the blocks that are smaller than the initial block size and makes the array size a nice power of two also.

TODO:

- Show equation for looking up the block for an array element index