Evaluating the Page Buffering and Cache Image Features

Mohamad Chaarawi

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. The HDF5 library’s metadata cache already does a reasonably good job of minimizing the metadata I/Os during normal operation. But whenever a flush of the metadata cache is triggered, the small I/Os will happen.

Two new features have been added to HDF5 to mitigate these small I/O operations: *Page buffering* will ensure that HDF5 I/O requests to a file can be minimized to a specific granularity and alignment, set by the application. The *cache image* feature, when enabled, will write the contents of the metadata cache to the file in a single I/O on file close, and then populates the cache with the contents of this block on file open – avoiding the many small I/Os that would otherwise be required on file close and re-open.

This document evaluates these two features by running a selection of benchmarks on several HPC systems with parallel file systems.

# Introduction

The page buffering feature adds a layer within the library immediately above the Virtual File Driver I/O layer, and acts 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 and is always aligned to a page boundary. 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.

Page buffering allows users to specify a maximum page buffer size to store page-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 the page; otherwise it reads in the page from disk, updates it, and caches it for future use. For more details, refer to the page buffering RFC.

The cache image feature serializes the contents of the metadata cache to an image on file close, instead of the usual processing in which dirty entries are written back to their assigned locations in the file and clean entries are discarded. Avoiding small metadata I/Os on file close and lumping them into a block, called the cache image, should increase the performance of file close on parallel file systems, since small write perform badly there. Consequently, on file re-open, the cache image is read back, to populate the cache to the same state it was before the last file close, instead of reading all the required metadata entries.

# Approach

We use two benchmarks for our evaluation of the page buffering feature:

1. H5perf\_serial: a benchmark included in the HDF5 distribution that tests write performance to a single HDF5 dataset using a configurable buffer size. In our results we vary the overall size and dimensionality of the dataset and the buffer and measure performance with and without page buffering. Using this benchmark, we also measure performance using the core VFD to validate the page buffering results, since the core VFD does all I/O in memory.
2. MACSio: a Multi-purpose, Application-Centric, Scalable I/O proxy application. MACSio is supposed to fill a long existing void in co-design proxy applications that allow for I/O performance testing and evaluation of tradeoffs in data model interfaces and parallel I/O paradigms for multi-physics, HPC applications. MACSio is under active development, and at this stage does not simulate a rich metadata access other than dataset and group creation. So the main focus will be on raw data access and the effect of page buffering.

The cache image feature is evaluated using MACSio. We use chunked datasets in our evaluation of the cache image to try to increase and scatter around the metadata in the file, but the amount of metadata is not going to be large and so we do not expect that the cache image will hugely affect performance on that front.

The results were gathered on two HPC supercomputers:

* Hopper at NERSC, a Cray XE6 system with a Lustre file system.
* Cetus at ANL, an IBM BG/Q system with a GPFS file system.

# Page Buffering Evaluation

In this section, we evaluate the page buffering feature using the benchmarks and system listed above.

## H5perf\_serial

The tables below show the results obtained by running the h5perf\_serial benchmark with one MPI rank on a compute node. The Dataset Size field indicates the size of the dataset being written to and its dimensions. Similarly the Access Size field indicates the size of the memory buffer and its dimensions. Note that when the access size is of multiple dimensions (1K x 1K), the HDF5 library internally translates that into multiple accesses of size equal to the slowest changing dimensions, so in this case, the I/O size without page buffering would be 1K. The PS / PB field indicates the size of a single page, essential the I/O access size if page buffering is enabled, and the total page buffer size indicating the total number of pages to keep in the page buffer respectively (0/0 means that page buffering is disabled).

The results gathered are the total bandwidth for writing and reading raw data (*H5Dwrite()* & *H5Dread()*), and total bandwidth including the file open and close operations. The bandwidth displayed is in MB/sec units.

### Hopper Results



A contiguous 1-D dataset and a write buffer of 1KB (IO size = 1KB).



A contiguous 2-D dataset and a 2-D write buffer with dimensions 1KBx1KB (IO size = 1KB).



A contiguous 2-D dataset and a 2-D write buffer with dimensions 1KBx1KB (IO size = 1KB).



A contiguous 3-D dataset and a 3-D write buffer with dimensions 1x1KBx1KB (IO size = 1KB).

### Cetus Results

Some of the Cetus results are marked with an x which means that for the particular configuration, a Cetus node does not have enough RAM to run the actual test to completion.



A contiguous 1-D dataset and a write buffer of 1KB (IO size = 1KB).



A contiguous 2-D dataset and a 2-D write buffer with dimensions 1KBx1KB (IO size = 1KB).



A contiguous 2-D dataset and a 2-D write buffer with dimensions 1KBx1KB (IO size = 1KB).



A contiguous 3-D dataset and a 3-D write buffer with dimensions 1x1KBx1KB (IO size = 1KB).

### Discussion

The results obtained on both systems show that there is a significant speedup with page buffering with several test configurations, especially with the small 2-D dataset of size 10Kx10K. In that case, the number of H5Dwrite() calls generated was not high compared to the other configurations. The speedup on the other configuration was still good, but not as significant. This prompted an investigation of where the time is spent since the I/O size with page buffering does not change. Further profiling showed that the actual I/O time is greatly reduced with page buffering, however other non-I/O operations in HDF5 turn out to be significant, skewing the total bandwidth reported above.

To facilitate this investigation, we profiled h5perf\_serial on Cetus with a 1-D dataset of size 300MBs and an I/O write buffer of size 1KB. This means that the benchmark generated 300,000 H5Dwrite() calls. We profiled with and without page buffering:



We measured the total time it took to complete the h5\_perf write phase (total h5\_perf\_write()) which sets up the I/O by setting the hyperslab selections and property lists, and calling H5Dwrite(). We measured the total time spent in the page buffer layer, which is just a pass through when page buffering is disabled, and the total time spent in the VFD layer which calls the lower level write operations to the file system. Finally we measure the time to close the file. Since the page buffer size used is larger than the dataset and the file metadata, no I/O happens when page buffering is enabled until the file is closed. Thus we conclude that the total I/O time is the time spent in the page buffer layer (copying the data into the cached pages) and the time spent in the file close (which includes the time spent in the VFD layer doing I/O when flushing the pages). On the other hand, when page buffering is disabled, the total time spent in I/O is the actual time reported from the VFD layer. So we can conclude that the I/O speedup here is:

215.2655 / (2.046 + 1.49) = ~60 X

The overhead of setting up the I/O itself (~20 sec) with and without page buffering is what skews the results in the previous results, since they are included when measuring the bandwidth. The I/O setup phase in HDF5 has now become the actual bottleneck, when page buffering is enabled; indicating that page buffering is actually working very well in reducing the I/O time. This exposed other places in the HDF5 library that need optimization, which wasn’t obvious when page buffering was not available.

## MACSio

MACSio provides several configuration parameters to vary the access pattern and I/O size of the applications. The results that we gathered using MACSio were all using the Multiple Independent File (MIF) mode, where the file is always accessed using serial HDF5 by one MPI rank, and that rank passes the baton to the next process in its group when it’s done. The following are the parameters that we varied with when testing MACSio:

avg\_num\_parts

part\_size (io request size)

vars\_per\_part

The results reported show the summed bandwidth resulting from all the processes running MACSio and the bandwidth for the time between the first starter and the last finisher, which we report in the tables as the “Pessimist” bandwidth.

### Hopper Results

Processes = 24, MIF = 1, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 2000:



Processes = 256, MIF = 8, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 2000:



Processes = 1024, MIF = 16, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 2000:



Processes = 8192, MIF = 64, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 3000:



Processes = 24, MIF = 1, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 256, MIF = 8, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 1024, MIF = 16, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 8192, MIF = 64, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 24, MIF = 24, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 4000:



Processes = 256, MIF = 256, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 4000:



Processes = 1024, MIF = 1024, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 4000:



Processes = 8192, MIF = 8132, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 4000:



### Cetus Results

Processes = 32, MIF = 2, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 500:



Processes = 256, MIF = 8, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 500:



Processes = 1024, MIF = 16, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 500:



Processes = 32, MIF = 2, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 256, MIF = 8, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 1024, MIF = 16, part\_size = 16384, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 32, MIF = 32, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 256, MIF = 256, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 500:



Processes = 1024, MIF = 1024, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 500:



### Discussion

The results on Hopper with the Lustre file system show that page buffering is yielding a speedup ranging from 1.2x to 3.8x with all the different configurations that we experimented with. It is worth noting there that the file system didn’t seem to be very sensitive to the small I/Os generated by MACSio. Almost all of the I/Os were from raw data (HDF5 datasets) that are contiguous on the file system, so the system itself might be doing optimizations by detecting our access pattern. The metadata generated by MACSio is relatively small, and usually all fits into a single page (1MB) when page buffering is enabled.

Further profiling on Hopper confirms our analysis. We profiled the total time required for different operations in MACSio and HDF5:

H5Fcreate : The time to create all the files.

write\_mesh: The time to setup the I/O (creating the property lists, dataspaces, dataset) and to do the I/O (writing to the dataset).

space+dcpl: The time to create the HDF5 dataset creation property list and the dataspace selections.

H5Dcreate: The time to create all the HDF5 datasets.

H5Dwrite: The time spent in the H5Dwrite operation

actual io time: The time spent in the H5Dwrite operation minus the internal time required to setup the I/O operation in HDF5. This should be almost equal to the time spent in the page buffer layer and the VFD layer doing I/O if any pages were evicted, or the total I/O time for raw data if page buffering is disabled.

H5Fclose: The total time to flush and close the file, including flushing all the pages from the page buffer if it is enabled.

Processes = 24, MIF = 1, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 500:



Processes = 1024, MIF = 16, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 1000:



The profiling results show that the actual I/O time and the file close time are relatively small in both cases with and without page buffering. Page buffering yielded 3.5-4x speedup of the I/O time. The results also show that the time spend in doing non-I/O operations such as creating datasets and setting up the I/O are significant compared to the actual I/O time and needs future investigation.

The results on Cetus show a bigger speedup when page buffering is enabled. The speedups with page buffering ranged from 2 to 24x. It was surprising to see that some of the results with page buffering enabled were still low, although they were better than with page buffering disabled. Profiling some of those runs show results similar to the one done with h5perf\_serial, where the non I/O operations in HDF5 were a big factor in reducing the calculated bandwidth:

Processes = 32, MIF = 2, part\_size = 4096, avg\_num\_parts = 3.5, vars\_per\_part = 500:



Processes = 1024, MIF = 16, part\_size = 4096, avg\_num\_parts = 4, vars\_per\_part = 1000:



We can see that the actual I/O time and the file close time are significantly improved when page buffering is enabled in both cases above. However the time to create the datasets and setup the I/O operations is very significant and those operations eat away at the overall bandwidth and the speedups recorded in the results from the previous section.

# Cache Image Evaluation

To evaluate the Cache Image feature, we use MACSio on Hopper and Cetus. We changed MACSio to create chunked datasets in an effort the increase the metadata size generated; however due to the nature of the tests, where each process creates and writes to different datasets and no processes open and access existing datasets, we expect the cache image feature to not yield a considerable performance benefit on that front. Loading the file metadata and the root group structures from the cache image block in one I/O is the main focal point here for performance benefit.

We measure the total time to complete all HDF5 calls, the sum for all the file create/open times, and the sum of all file close times on all MPI ranks.

## Hopper Results

Processes = 32, MIF = 1, part\_size = 80000, avg\_num\_parts = 4, vars\_per\_part = 200:



Processes = 32, MIF = 1, part\_size = 80000, avg\_num\_parts = 4, vars\_per\_part = 1000:



Processes = 32, MIF = 1, part\_size = 80000, avg\_num\_parts = 20, vars\_per\_part = 1000:



Processes = 256, MIF = 4, part\_size = 80000, avg\_num\_parts = 4, vars\_per\_part = 200:



Processes = 256, MIF = 4, part\_size = 80000, avg\_num\_parts = 4, vars\_per\_part = 1000:



Processes = 256, MIF = 4, part\_size = 80000, avg\_num\_parts = 20, vars\_per\_part = 1000:



## Cetus Results

Processes = 32, MIF = 1, part\_size = 80000, avg\_num\_parts = 4, vars\_per\_part = 200:



Processes = 32, MIF = 1, part\_size = 80000, avg\_num\_parts = 4, vars\_per\_part = 1000:



Processes = 256, MIF = 4, part\_size = 8192, avg\_num\_parts = 4, vars\_per\_part = 200:



## Discussion

The results obtained on Hopper indicate that cache image has little effect on the performance. In some cases we see that the performance drops with cache image turned on. This can be attributed to the fact that we are reading extra metadata that is not needed from the cache image in every file open and this large block read was not fast enough to offset the small reads of the needed metadata when cache image is turned off on this particular system. We expect this to be different with applications that open the file and accesses more existing metadata.

On Cetus, the results seem more favorable to the cache image feature. We see 2-3x speedups in file close time with cache image. This indicates that writing the metadata in one block is better than writing the metadata one entry at a time. Note that this does incur an extra overhead at file open with cache image since we are reading some metadata that we are not using in the above scenario, but that overhead seems negligible compared to the time we save on file close.

# Summary

We evaluated the new page buffering and cache image features on two HPC systems. The results obtained for page buffering using two benchmarks show good speedups over non page buffered runs. Profiling some configurations of the benchmark runs also indicate that there is a considerable non I/O overhead incurred by the HDF5 library and needs to be investigated and addressed in the future.

We did not have a very suitable benchmark to evaluate the cache image feature. Using MACSio for this evaluation shows a glimpse of the optimization we can get on file close on Cetus, however, to fully leverage this feature, a metadata rich application needs to be evaluated instead.

Both the page buffering and cache image features will be further tuned and improved to be included in the HDF5 1.10 release in 2016. Beta releases of the 1.10 release will be available sooner and we expect that those features will be included there too.

# Revision History

|  |  |
| --- | --- |
| *September 28, 2015:* | Version 1 circulated for comment within The HDF Group. |