Sparse Data Management in HDF5

John Mainzer[1], Neil Fortner[1], Gerd Heber[1], Elena Pourmal[1], Quincey Koziol[2], Suren Byna[2], Marc Paterno[3]

[1]The HDF Group,
[2]Lawrence Berkeley National Laboratory
[3]Fermi National Accelerator Laboratory

{mainzer, nfortne2, gheber, epourmal}@hdfgroup.org,
{koziol, sbyna}@lbl.gov, paterno@fnal.gov

The importance of sparse data management is growing with data produced by large-scale experimental and observational facilities that contain small amounts of non-zero values. In this document, we explore different design options to support sparse data in HDF5, one of the most popular high-performance I/O libraries and file formats used for scientific data. We discuss several use cases and requirements. Our main *hard* design constraint is that any changes to the HDF5 dataset API would be a burden on users and not acceptable. The remaining options are discussed below. We provide rationale for what we believe is the strongest candidate and describe how its potential benefits can be simulated with the existing HDF5 library. We have conducted a set of computational experiments the results of which are reported here. They show that our candidate meets all relevant requirements and gives us a certain degree of confidence for an HDF5 library-native implementation.

Index Terms—**Sparse Data Management, HDF5**

I. INTRODUCTION

HDF5 is designed to store and manage high-volume and complex science data, and has become the leading I/O middleware solution and file format. HDF5 allows storing generic data objects within files in a self-describing way, and much of the power of HDF5 stems from the flexibility of the objects that make up an HDF5 file: *datasets* for storing multi-dimensional arrays of homogeneous elements and *groups* for organizing related objects. HDF5 *attributes* are used to store user-defined metadata on objects in files. Due to the simplicity of the HDF5 data model and flexibility and efficient I/O of the HDF5 library, HDF5 supports all types of digitally stored data, regardless of origin or size. Petabytes of experimental and observational data (EOD) produced by remote sensing data collected by satellites, high-energy physics experiments, and MRI brain scans are stored in HDF5 files, together with metadata necessary for efficient data sharing, processing, visualization, and archiving.

Despite the widespread use of HDF5 for storing EOD, there is no concept of data *density* or *sparsity* in the HDF5 data model. Problem-sized data is represented as HDF5 datasets which can be thought of as multidimensional, logically "dense" (as opposed to ragged!), rectilinear arrays of a certain element type with a corresponding default or fill-value.

The implementation provided in the HDF5 library lets users control several aspects of how datasets are mapped into HDF5 files, such as layout (e.g., contiguous or chunked), allocation time (early or late), and storage initialization (fill-value if set, or never). Given this implementation, there are two obvious approaches to representing sparse data in HDF5:

1) Use chunked layout and, optionally, compression
2) "Condense" the data and mimic a sparse storage format

The most troublesome side-effect of the first approach is that it is impossible to efficiently determine which values of a sparse dataset are *defined*, i.e., represent "non-zero" or non-default values[1]. The loss in I/O performance due to the overhead introduced by (de-)compression is another potential downside. On the upside, the first approach maintains the abstraction or the logical view of the dataset. The second approach "destroys" that abstraction, which is its most problematic side-effect. It forces users to adopt an additional protocol or API whose sole purpose is to "undo the damage".

While both approaches have a place under the right circumstances, their side effects limit more general use. In this paper, we explore the options for and feasibility of sparse data management in HDF5 *without changes to the existing* API. We begin with the description of a model problem (§II). This is followed by a fairly comprehensive discussion of design options with their pros and cons (§III). The winning design is then applied to the model problem and simulated to obtain estimates of its potential space and run-time savings (§IV).

II. A MODEL PROBLEM

There is no shortage of sparse data use cases in many scientific disciplines and experiments. In preparing this paper, we drew on our collective experience with applications from High Energy Physics (neutrino detection, trajectory data analysis, accelerator modelling), light sources (neutron and X-ray scattering experiments), mass spectrometry, and compressive sensing. The intended readership of this document should also have no difficulty to add plenty of their own examples.

The following model problem is based on an experimental physics use case [1]. Assume a stream of large (1–4 MP),

---

[1]HDF5 library version 1.10.5 contains an API to determine which chunks of a dataset are actually allocated, but that's far from finding defined entries efficiently.

two-dimensional data sets (images) arriving at frequency f, where the dimensions of the images are constant over time. Further assume that for each image, it is possible to identify automatically either (see figure 1):

1) A rectangular *Region Of Interest* (ROI) which will typically comprise about 10% of the 2D dataset, or
2) 50-100 *small clusters* of pixels, irregularly shaped —say 5-10 contiguous points or pixels.

The ROI shape and position, and the number, size, shape, and location of the small clusters will change over time. For each image, store only the ROI or the clusters in a 3D dataset (the third dimension is the "time" index of the 2D dataset in the stream). However, we must be able to recover both the location and contents of the ROI or the clusters.
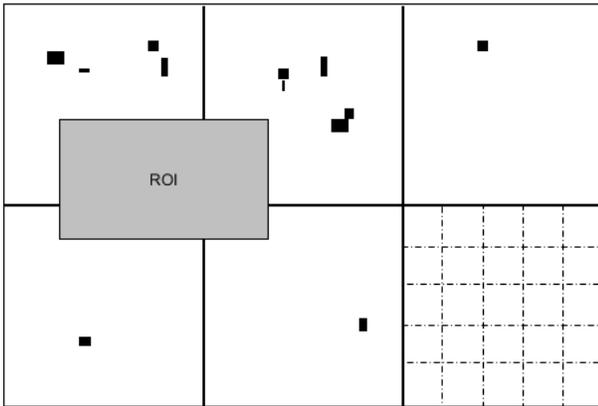

Figure 1: A 2D dataset with an ROI and several small clusters of non-zero data.

In addition, for some $n \geq 1$, store every n-th 2D dataset in full. (n is constant with typical values between 1 and 10,000). This could be done either in the 3D dataset mentioned above, or in a separate 3D dataset.

The overall objective is to *drastically* reduce the quantity of data stored by discarding the "uninteresting" parts. By storing every n–th 2D dataset in full the correctness of the automatic recognition of ROIs or point lists can be verified.

A. Requirements

1) The HDF5 representation should preserve the (sparse) subset in (dense) superset abstraction.
2) It should allow access to sparse datasets with the *existing API* (albeit with inefficiencies in some cases).
3) The representation should yield commensurate reductions in storage space and I/O time.
4) Arbitrary (random access) read and write operations must be supported. If possible, provide optimizations for patterns such as WORM (write-once-read-many).
5) The representation must allow data-parallel operations.

To satisfy requirement 2 would most likely satisfy requirement 1. It would also have the benefit of being the least disruptive to the existing HDF5 ecosystem. There is no question about the value of a good abstraction, but it might

be difficult to measure. Requirement 3 fills that "gap." Finally, even the largest sparse dataset may not fit all at once in a fast layer of the memory hierarchy, and we need to ensure that it can be "divided and conquered" regardless.

III. DESIGN OPTIONS

The first comprehensive discussion of design options was given in [1] where two categories of options were identified: Either an attempt is made to stretch the existing dense design (*weak options*) or sparse data are treated as fundamentally different (*strong options*).

A. Weak Options

a) Status quo: Make do with what's in HDF5 today.

Pros: This approach should perform reasonably well with chunked dataset layout for the ROI part of the use case. Only overlapping chunks will be allocated and the 10% area coverage of an ROI means that one wouldn't need a huge number of tiny chunks.

Cons: The small cluster/point list part of the use case poses a serious problem. Since the regions of a chunk that are populated by cluster elements are no longer rectangular it is almost impossible to keep track of defined ("non-zero" in sparse matrix terminology) entries unless an extremely small chunk size is chosen, which would inevitably lead to a bloated chunk index and the overhead of many small I/O operations.

b) Chunked datatset with fill-value filter: Use a filter to save space by not storing fill-values.

Pros: This approach fits w/ the current infrastructure and delivers the space savings *on disk*.

Cons: There are no space savings in memory. It is also unclear how to efficiently determine which values on a chunk are not fill-values. Both deficits could be addressed by storing selection information alongside the filtered chunks, but would require substantial changes to the dataset API (`H5D`) and the storage format. There is also the potential for "disruption" of the filter pipeline such as limitations on interoperability with other filters.

c) High-level library: Use the existing primitives to store the "condensed state" of sparse structures and implement a high-level API that maps between the (logical) sparse view and the sparse format encoded as multiple (dense) HDF5 datasets.

Pros: This approach supports the use case and can deliver the space and time savings required. Staying outside the core HDF5 library simplifies maintenance.

Cons: While this approach maintains the level of abstraction for library users, it is abstraction defeating to the rest of the ecosystem. Substantial changes to existing tools and applications would be required. It might work for a class of use cases, but it is unclear how it could be generalized. Non-WORM data and updates would be difficult to support, especially in a parallel setting.

B. Strong Options

a) B-trees: Store defined or non-fill-value entries in B-trees indexed by their (logical) position.

Pros: The use of B-trees in the HDF5 library is pervasive, and the re-use of existing machinery would be an advantage. This approach would deliver the space and time savings for small clusters.

Cons: The overhead for "not-so-sparse" data would be considerable. There are also better options (trees) available. Finally, a parallel implementation of the tree manipulations and metadata management would be rather challenging.

b) Sparse chunks: Combine chunks and selections. A *sparse chunk* is a chunk "decorated" with a selection that marks the positions of non-fill values.

Pros: We re-use existing selection machinery. It delivers space savings on disk and in memory, as well as time savings. Most importantly, it is abstraction preserving.

Cons: The potential semi-sparse data overhead needs to be estimated. The optimization of complex selections is a familiar challenge.

c) Start from scratch:

Pros: Everything's perfect here.

Cons: (None.) High risk, uncertain outcome and cost.

C. Discussion

Let's identify the strongest contender in each category! Among the weak options, if we reject the status quo, only two options are left. The fill-value filter approach cannot reduce the in-memory footprint of sparse datasets, and the efficient determination of defined entries requires additional metadata structures. The main downside of the high-level library is its abstraction defeating nature and the knock-on effect on the ecosystem. However, it is clearly the approach one would pursue with today's HDF5 for a *specific* use case.

Among the strong options, it's safe to reject the from-scratch option. The main issues of the B-tree approach are the challenging parallelization and user complexity (B-tree parameters). The sparse chunks approach, by comparison, is based on well-established user-level constructs (chunks, selections) and offers chunk-level parallelism at the minimum.

There is a certain correspondence between the high-level library approach (HLL) and sparse chunks (SC). By similar means, HLL might achieve outside the HDF5 core library what SC can achieve inside the library. SC's main advantages are: 1) It is abstraction preserving. 2) Being inside the library, it is application-neutral. 3) Optimization is possible w/o disturbing the ecosystem.

How about backing up this analysis with a few empirical results? The "duality" between HLL and SC permits us to do exactly that. In the next section, we describe an experiment whose goal is to make it plausible that the space and time savings goals are indeed achievable (for our model problem).

For a study of the HDF5 library changes necessary to implement sparse chunks we refer the reader to [2].

IV. EXPERIMENTAL EVALUATION

The conclusion of the options discussion in section III-C was that we can simulate the potential benefits of the sparse chunks design by implementing a high-level library-like I/O kernel for our model problem. In this section, we describe such an implementation and report the results obtained.

A. Four I/O Kernels

We implemented four I/O kernels for our model problem.

Baseline: Write a single 3D dataset of full 2D frames, and read it back.

ROI: At a constant rate, write a reduced number of full 2D frames (key frames). In all other, non-key frame time steps write an ROI covering 10% of the full frame area where the coordinate range covered by the ROI is represented as an encoded HDF5 hyperslab selection. Subsequently, read back the key frames and the ROIs (in chronological order).

SC (Small Clusters): At a constant rate, write a reduced number of full 2D frames (key frames). In all other, non-key frame time steps write a random number (between 50 and 100) of small hyperslabs containing at least 5 to 10 contiguous pixels which are represented as encoded HDF5 hyperslab selections. Subsequently, read back (in chronological order) the key frames, and the small clusters.

Subsets (SS = ROI|Small Clusters): Write a reduced number of full 2D frames (key frames). In all other, non-key frame time steps, flip a coin and either write an ROI covering 10% of the full frame area or write a random number (between 50 and 100) of small hyperslabs containing at least 5 to 10 contiguous pixels. Both are represented as an encoded HDF5 hyperslab selections. Subsequently, read back (in chronological order) the key frames, and the subsets (ROIs or the small clusters).

Without loss of generality, we can represent the small clusters of pixels as unions of (small) rectangles. According to the LCLS-II specification, an ROI covers about 10% of pixels. By contrast, the small clusters in a non-key frame time step cover only about 0.1% of pixels. It could be argued that by making the chunks smaller, the case of small clusters (SC) is a mere re-scaled version of ROI. However, the two cases exhibit very different ratios in the amounts of metadata (bookkeeping) to data (pixel colors). The ratio is close to zero for a rectilinear region that covers about 10% of pixels, but it'll be significant for 50-100 small clusters whose pixel coverage is a mere 0.1% of pixels. This is illustrated by the third kernel (SC). The fourth kernel (SS) doesn't follow the LCLS-II use case verbatim, but is a "blend" of 50% ROI and 50% of SC. It offers a glimpse of the impact of the metadata to data ratio on performance.

The kernels can be parameterized as follows with the values used in the experiment shown in parentheses:

B - the size of a pixel (color depth) [bytes] (1, 2, 4)
N - the logical number of frames (128)
W - the width of a frame [pixels] (1024)
H - the height of a frame [pixels] (1024)
S - the stride between full-frame writes (16)

For each kernel we record the following metrics:

- The size of the output HDF5 file in bytes.
- The amount of application metadata read/written.
- The amount of application data read/written.
- The total application metadata read/write time.
- The total application data read/write time.

By *application data* we mean pixel colors of depth B. The ROI and small cluster encodings as well as other book-keeping datasets are considered *application metadata*. By comparing the sizes and timings against the baseline (full frames!), we obtain an estimate for the potential space/time efficiency of the sparse chunks design.

B. Implementation

The I/O kernels were implemented in C++ 17 with help from two third-party libraries. We used Armadillo – a C++ library [3] for linear algebra for & scientific computing – to represent 2D frames and vectors in memory. We also used H5CPP [4] – a header-only C++ template library for HDF5 – with its convenient and efficient packet table implementation.

Although a contiguously stored 3D dataset of full images is not realistic for the use case at hand (since the number of full frames to be written is unknown), it makes for a good *baseline* with its performance close to standard POSIX I/O. On the other hand, HDF5 1.8.11 and 1.10.2 saw the introduction of direct chunk write (H5Dwrite_chunk) and read (H5Dread_chunk) functions, respectively, to bypass the default I/O path for chunks. We confirmed that the performance of direct chunk I/O against an arbitrarily extendible dataset is equivalent to the read and write performance against contiguous layout. By comparison, the performance of H5Dwrite along the default chunked I/O path was poor while the performance of H5Dread was acceptable. Consequently, we used direct chunk I/O in the implementation of all kernels. Unlike for full frames, the by comparison much smaller I/O requests and size variability required the introduction of buffering (for writes) and prefetching (for reads). This can be seen as mimicking the function of a cache manager in a real sparse chunks implementation.

C. Results

The following results were obtained on `cori` at NERSC and `jelly` at The HDF Group. We used GCC/G++ 7.3.0 (-std=c++1z) and HDF5 1.10.4 with the latest file format. The frames were of a width and height of 1024 pixels. The logical number of frames was 128 with a stride of 16, and each kernel was run 10 times.

Each `cori` node has two sockets and each socket is populated with a 16-core Intel Xeon Processor E5-2698 v3 ("Haswell") at 2.3 GHz. Each node has 128 GB DDR4 2133 MHz memory (four 16 GB DIMMs per socket). Each core has its own L1 and L2 caches, with 64 KB (32 KB instruction cache, 32 KB data) and 256 KB, respectively; there is also a 40-MB shared L3 cache per socket. The test were run against the scratch Lustre file system on a Cray Sonexion 2000 Lustre appliance with a maximum aggregate bandwidth > 700 GB/sec. (The use of the Burst Buffer on `cori` was

not considered for this experiment. We would expect better performance than on Lustre. However, the use of a local file system on `jelly` should give us fairly accurate picture of what, if any, qualitative differences to expect.)

`jelly` has two sockets and each socket is populated with a 14-core Intel Xeon Processor E5-2695 v3 at 2.3 GHz. It has 125 GB DDR4 memory. Each core has its own L1 cache and L2 caches, with 32 KB and 256 KB, respectively; there is also a 35-MB shared L3 cache per socket. The tests were run against an XFS file system over a RAID 10 volume which consisted of 4x Seagate ST6000NM0034 (6TB) drives managed by a LSI Logic / Symbios Logic MegaRAID SAS-3 3108 [Invader] controller.

We report the space "compression" and time "speedup" factors of the ROI, SC, and SS I/O kernels relative to the baseline of reading/writing full frames from/to a contiguous 3D dataset. Given the baseline time T^b and I/O kernel timings t^k_D, t^k_{MD} for kernel k and (meta-) data (D, MD), the time speedup factor was calculated as $T^b/(t^k_D + t^k_{MD})$. Similarly, given the baseline file size S^b and the kernel k's output file size of s^k, the space compression factor was calculated as S^b/s^k. A factor of one means that the kernel didn't save ("compress") time or space over the baseline. A factor less than one means a slowdown or bloat over the baseline (or both).

The I/O kernel names are 'ROI', 'SC', and 'SS' (subsets = ROI|SC). The number after a kernel name indicates the pixel size in bytes (1, 2, 4). For example, 'ROI2' indicates an ROI kernel run with a 2 byte pixel size (16-bit "color depth").

All results shown below are relative to the baseline of writing to/reading from a three-dimensional contiguous dataset of full frames. For reference, the absolute performance numbers for `jelly` and `cori` are shown in table II.

TABLE I: Size of the baseline HDF5 file.

Color Depth [byte]	File Size [byte]
1	134,219,776
2	268,437,504
4	536,872,960

TABLE II: Baseline performance `cori` (1) and `jelly` (2).

Mode	Color Depth [byte]	Time 1 [us]	Time 2 [us]
WRITE	1	232,057	58,134
READ	1	28,150	40,849
WRITE	2	456,126	134,617
READ	2	59,658	77,796
WRITE	4	741,495	269,834
READ	4	108,527	134,485

Because of space constraints, only a selection of figures is shown below. Overall, the results for both systems exhibited the same qualitative behavior.

D. Discussion

The results seem to be consistent with respect to their baselines across platforms.

For ROI (see figure 2), the time speedup factor for writes and reads appears to be between 6 and 10. Similarly, the space
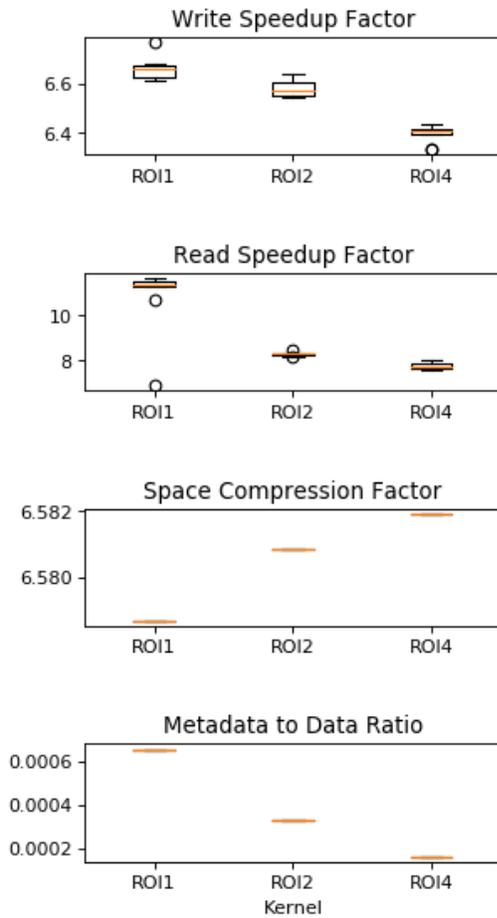
Figure 2: Space and time "compression" for ROI. (`cori`)

Figure 3: Space and time "compression" for ROI, 8-bit color, and 10% coverage increments. (`jelly`)

compression factor for ROI appears to be around 6.6, and the metadata to data ratio is close to zero (< 0.001).

In figure 3, the effect of varying the coverage ratio in 10% increments of an ROI is shown. As expected, read, write, and space factors decline with increased ROI coverage. Higher color depth has a "calming" effect: In the corresponding figure for 4 byte colors (not shown here!) the factors are greater and the decline to 1 is more gradual.

For SC (see figure 4), because of the unfavorable metadata to data ratio, color depth is a significant factor. (The ratio is greater than 2 for 1 byte color depth and even for 4 byte color depth the ratio is still close to 1.) The variability in the factors is in the 5 to 10 range, at a level comparable to ROI.

For SS (see figure 5), the time factor decreases to between 4 and 6 and the space compression decreases to about the same range. The metadata to data ratio jumps to between 0.2 and close to 1. The "calming" effect of the color depth can be seen clearly in this case, as greater color depth creates larger data writes and reads while the metadata reads and write remain unchanged.

Overall, color depth appears to be much less of a factor in the ROI case than for SC and SS, which is plausible. While these numbers don't prove anything they suggest that
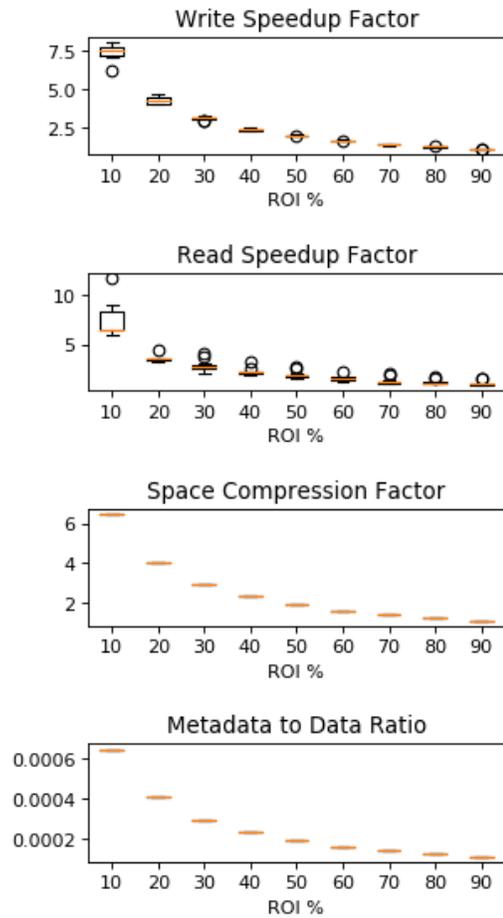
requirement 3 (see section II-A), which asks for commensurate reductions in space and time, is achievable in this and similar use cases.

V. RELATED WORK

The sparse chunks design resembles in some aspects the data tiles used in TileDB [5]. In HDF5, when creating a dataset with chunked layout a chunk size must be specified. Similarly, in TileDB, when creating a sparse array, a user can specify the data tile capacity. The idea of using HDF5 selections to track defined entries on a chunk mirrors TileDB's coordinate materialization approach for sparse arrays. The two approaches begin to deviate when it comes to which operations they support efficiently. As demonstrated in [5], TileDB's avoidance of in-place updates offers superior random write performance and handling of variable-length data. While a similar effect can be achieved in HDF5, for example, through the use of a so-called "onion" VFD, historically, this type of operation has not been a prominent use case. These use cases might become more important if HDF5 were to support sparse data through sparse chunks or another mechanism. It also increase the demand for concurrent, atomic read/write access to HDF5
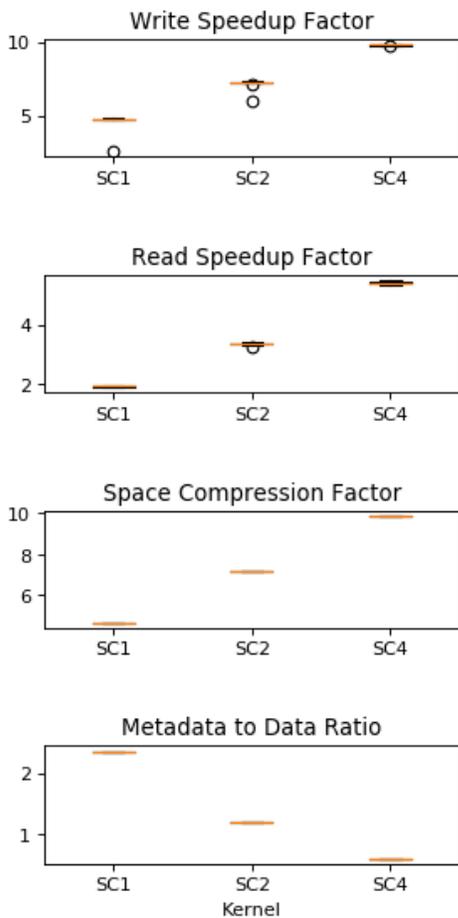
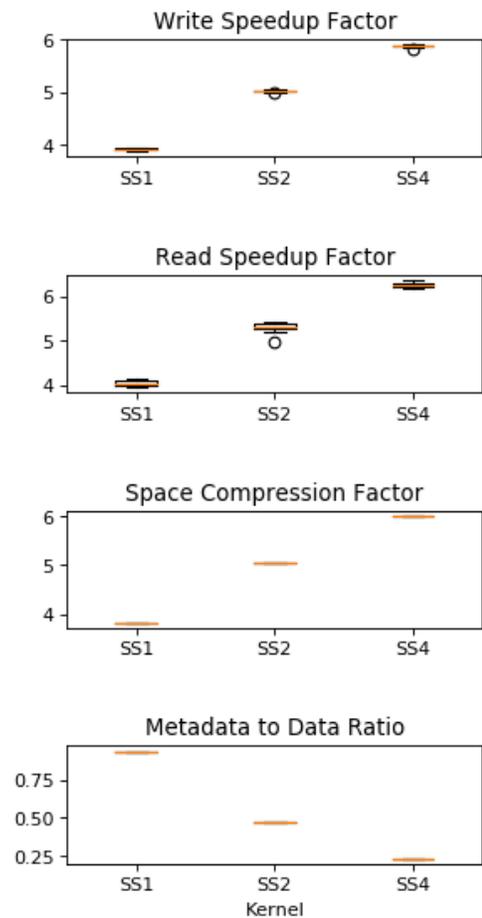Figure 4: Space and time "compression" for SC. (`cori`)

Figure 5: Space and time "compression" for SS. (`cori`)

datasets, which is another area where TileDB's approach of fragments (timestamped snapshots) pays off.

VI. Conclusions

Inspired by a "groundswell" of use cases, in this paper, we have explored design options for the support of sparse data in HDF5 subject to the requirements described in section II-A and without changes to the existing API. We have provided the rationale for and chosen the option in our judgment most sensible candidate. We then evaluated this approach using a model problem and an HDF5 library-*external* prototype. Our findings show that it meets the requirements for the particular setup and we see this as a confirmation of the suitability of our design choice.

VII. Future Work

There is no other "proof" of our design than "to build the real thing". A preliminary study [2] shows that the complexity of the first version of an HDF5 sparse chunks implementation is comparable to other HDF5 features. Our near term goal is to get this analysis into the hands of the HDF community and recruit supporters and early adopters. A revised request for comments (RFC) document will be published this fall (2019) and implementation planning could begin as early as Q4 2019.

References

[1] J. Mainzer, "Sparse dataset design options," *HDF internal*, May 2018.

[2] N. Fortner and J. Mainzer, "Rfc: Sparse chunks," *HDF internal*, August 2018.

[3] "Armadillo c++ library for linear algebra & scientific computing," http://arma.sourceforge.net/, accessed: 2019-02-05.

[4] "H5CPP easy to use c++ templates for hdf5," http://h5cpp.org/, accessed: 2019-02-05.

[5] S. Papadopoulos, K. Datta, S. Madden, and T. Mattson, "The tiledb array data storage manager," *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 349–360, Nov. 2016. [Online]. Available: https://doi.org/10.14778/3025111.3025117