HDF5 for high data rate synchrotron and x-ray free electron laser applications

Eugen Wintersberger

November 21, 2011

1 Motivation

Today, research facilities, using synchrotrons and/or free electron lasers (FEL) as x-ray sources, are facing tow major problems concerning data formats:

1. the abundant amount of different binary file formats for detector data
2. and the high rate at which this data is generated.

The former problem has historical reasons. When 2D area detectors became more popular, research facilities and detector vendors quickly realized that, due to the large amount of data produced by these detectors, writing such data to ASCII files would not be possible within reasonable time. Consequently each detector vendor or lab developed its own binary file format (EDF [1], CBF [2], etc.) or abused an existing format like TIFF[1] to store detector data. Most of this formats are incapable of storing additional meta-data along with the actual detector data. Hence many detectors write an additional output file (usually ASCII) in order to store supplementary data required for data analysis. To make data availble to users programs must be provided to read all these various data formats. The acquisition of a new detector usually goes along with new code to support its file format. Additional efforts are required to maintain all the code.

The second problem, high data-rates, is rather recent. At some point in time scientists discovered the beauty of time-resolved experiments. Unfortunately many physical effects of interest occur on a very short time scales. Thus, the rate at which detector data is recorded during a single experiment increased dramatically over the last years. These high data-rates have some unpleasant consequences. The first is the large number of files created during an experiment. Most of the file-formats used today for detectors are not capable of storing more than one detector image per file and if they are this feature is hardly used[2]. Along with meta-data files a single experiment can easily produce 100000 files. Such large numbers of individual files takes nearly every file-system to its limits. Aside form the sheer number of files also the total amount of storage consumed by the data becomes problematic. While in the early days a single experiment run occupied some MBytes of data, todays setups create hundreds of GBytes which must be stored for evaluation and archived later.

It is one of the goals of the German HDRI [3] project to find solutions for these problems by

1. agreeing on a common data format for all facilities contributing in the project
2. and developing technologies to store data using this format at high data-rates.

The project members have chosen HDF5 as the basement for our file format. While HDF5 provides us with everything we need to organize data in a single file (including meta-data), semantics will be added to the various groups and data-sets using NeXus [4]. Although HDF5 provides basically everything we need some problems remain in particular when it comes to high data-rates. These issues shall be discussed in the ongoing sections.

[1]This lead to a large variety of dialects which cannot be read or processed by standard software.

[2]TIFF is a good example for this problem. Though being able to store image stacks most detector systems write only a single image per file.

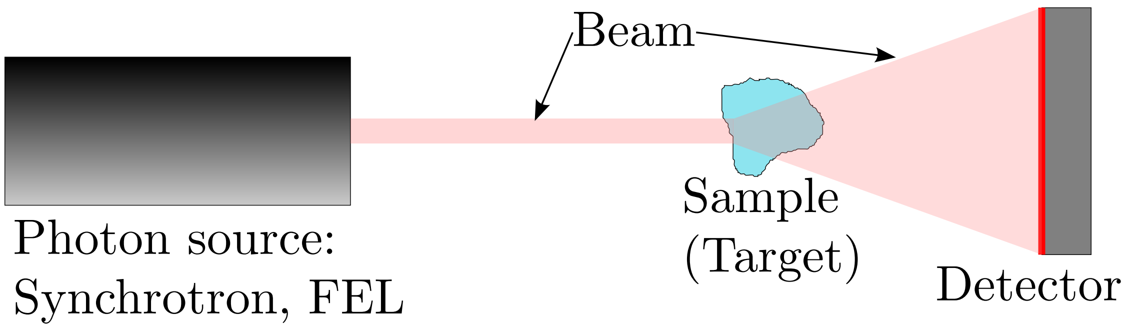

Figure 1: The basic setup for a synchrotron or free electron laser experiment. A beam of photons emitted by a source hits a sample and the scattered and/or transmitted photons are recorded by a detector.

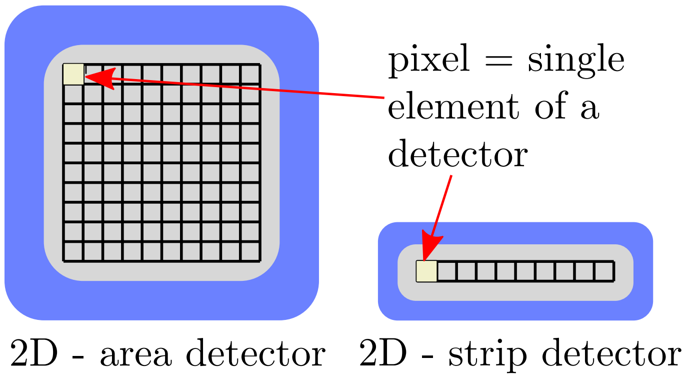

Figure 2: Two types of detectors are relevant for our HDF5 applications: 1D strip and 2D area detectors. The active (photon counting) elements in each of the two detector types will be referred to as *pixels*. Each pixel corresponds to a single emement in a multidimensional dataset.

2 Introduction

Figure 1 shows a simplified setup for an synchrotron radiation (SR) or FEL experiment. Light (photons) is emitted by a source shown on the very left of this sketch. The photons hit the sample (also called target) and the transmitted and/or scattered light is recorded by a detector placed in the vicinity of the sample. Two major types of detectors are currently in use: 1d strip and 2d area detectors (see Fig. 2). Point- or 0d-detectors exist but are not of relevance for high data-rate applications and thus will not be considered here. Detectors record data by counting the number of photons impinging on each of their active regions. These active (photon counting) regions of a detector (for both, 1d and 2d detectors) will be referred to as *pixels*. At each data point of the experiment the detector is exposed to the radiation for a particular time called *exposure* or *counting time* τ_c. During this time the detectors' pixels accumulate data (count photons). After the counting time data acquisition by the detector is stopped and read out by the acquisition time. The set of pixels values will be referred to as *frame* or *image*. Each of this frames is stored to disk or whatever storage medium is in use. After an experiment with N data points one usually ends up with N detector images.

2.1 Data organization in the HDF5 file

Instead of storing each frame in an individual file (as it is done now) we want all frames of an experiment to be stored in a single HDF5 data-set. Figure 3 shows the planned data-set layout for a 1d and a 2d detector. The rank r of the data-space is given with $r = 1 + d_{\text{detector}}$ where d_{detector} is the rank of a single detector frame ($d_{\text{detector}} = 1$ for a strip and $d_{\text{detector}} = 2$ for an area detector). The maximum shape $s_{\max}$ of such a data-set is given with $s_{\max} = (\texttt{H5S\_UNLIMITED}, n_1, \ldots, n_{d_{\text{detector}}})$ where the n_i denote the number of pixels along each detector dimension. The first dimension of the data-set corresponds to the number of points recorded during an experiment. Setting the maximum size of a dimension to `H5S_UNLIMITED` allows continuous

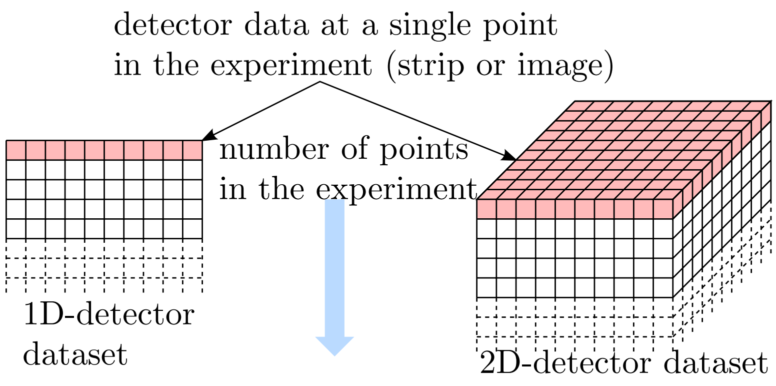

Figure 3: Organization of experiment data for 1D and 2D detectors. The first dimension of the data-set runs over the points at which data is recorded. The other dimensions correspond to the size of the detector (number of pixels along each dimension).

Name	Remark	CPU	RAM	Harddrive
PC1	the authors home desktop	Intel Core i7-956	12 GByte PC1333	1 TByte SATA3
PC2	standard DESY workstation	Intel Core 2Duo E8400 (3 GHz)	8 GByte PC1333	500 GByte SATA2
PCE	workstation for a Perkin Elmer detector			
PCP	DECTRIS Pilatus control system	Intel Xeon E5504	16 GByte PC1333	NFS over 10GBit FC

Table 1: Description of systems used for testing the *Names* mentioned in the table are used to refer to the systems in the text below.

expansion of the data-set along this dimension making it easy to append additional data points. In order to keep the data-set extensible and use compression, a chunked layout will be used. The shape of a single chunk is given with $s_{\text{chunk}} = (1, n_1, \ldots, n_{d_{\text{detector}}})$. Thus, each chunk represents a single detector frame.

As detectors basically count photons the data type used to store detector data will, in most cases, be an unsigned integer type. Typical sizes are 8, 16, and 32 bit. However, novel detector systems may also show other sizes like 4, 12, and 14 Bits.

2.2 A more abstract point of view on these data structures

From a programmers point of view, in the above approach, the data-set can be considered as a container (like a list or vector in C++) holding data objects of a particular type and shape. The first dimension of the data-set represents the container index of an element. As mentioned previously the data-sets can be extended along the dimension of size `H5S_UNLIMITED`. This feature will be used in a very excessive way: when a data-set is created its initial size along the extensible dimension will be 0 indicating an empty container. Storing data means nothing else then appending data to such an container. Each time a new detector frame is written the data-set is extended by a single chunk. This procedure allows a system to write data without knowing the final number of points being written to the file. Although the user usually specifies the number of points in an experiment before starting, there are situations like aborting an experiment which will render this number invalid.

3 Benchmarks

It is impossible to choose a file format for high performance applications without benchmarks. Hence we have performed several test with HDF5 in order to verify whether or not is is appli-

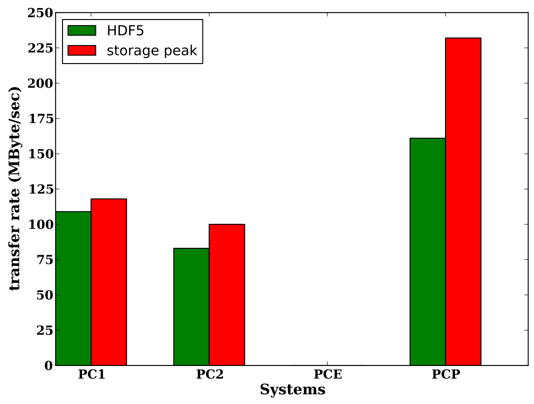

Figure 4: Comparison between HDF5 and storage device peak performance for 1024 × 1024 `unsigned short` detector frames.

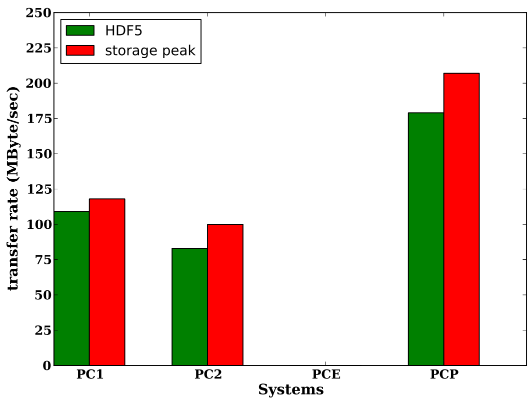

Figure 5: Comparison between HDF5 and storage device peak performance for 4096 × 4096 `unsigned short` detector frames.

cable for our high data-rate detectors. As shown in Tab. 1 common PC hardware was used for the benchmarks. From the point of processor and memory all systems used are pretty common PC hardware. Only the persistent storage devices are different for the different systems. Only hardware systems used at DESY where involved in the tests.

This section presents the most important results from our benchmarks showing that HDF5 in general satisfies our needs. However, some improvements should be made to make HDF5 ready for future applications. The code for the benchmarks is straight forward without special tweaks or tricks. As sample data for the tests we assumed a 2D area detector using a data-set layout as shown in Fig. 3. The data type was written was of type `unsigned short`. The peak performance of the used storage devices was measured by writing a 21 GByte file to the device using the `dd` command line tool. The block size used for `dd` was set to the size of a single detector frame.

3.1 Write to disk, RAID, and network storage

The first benchmark presented considers write-operation to conventional persistent storage devices including hard-disks (CP1, PC2), RAID systems (PCE), and network storage (PCP). On each device presented here a 21 GByte file was written with detector frames of sizes ranging from 1k × 1k to 4k × 4k without compression. Figures 4 and 5 show the results for the 1k and 4k frames respectively. On all storage systems HDF5 was able to achieve nearly device performance. It follows from a comparison of the PCP columns in Figures 4 and 5 that the frame size is only important for the NFS system. This is a rather important information. It is yet not clear whether it would be a good idea to increase the chunk size for small detectors or to tune the NFS system to get optimum performance for small detectors.

3.2 Write to RAM-disk

Although not being persistent, the ultimate high-performance storage device is a RAM-disk. Thus, RAM-disks are a suitable temporary storage device for experiments where the data-rates become too high for any other device available. Figure 6 shows the results from the benchmarks in comparison with the measured performance of the RAM-system. The problem here is to decide which performance marker is of importance. The red bars in Fig. 6 show results obtained from the STREAM benchmark measuring raw RAM performance. However, we are more interested in the performance of the `tmpfs` partition (denoted by the blue columns in Fig. 6) onto which the data was dumped. The `tmpfs` write-rate was measured using `dd`. If we compare now HDF5s' data-rate with the one obtained from `dd` we are back at device performance.

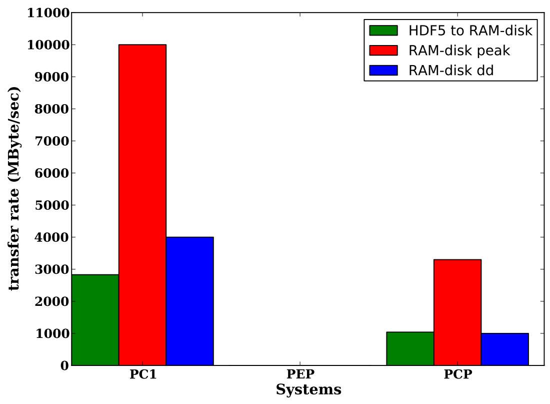

Figure 6: Benchmark on RAM-disks on various testing systems. Here the write performance drops significantly below the devices' capabilities. One reason might be that the data has to be copied from one location in memory to another. However, the HDF5 performance is close to what we can achieve with dd on a RAM-disk, and this is maybe the more reasonable number for the performance of a RAM-disk.

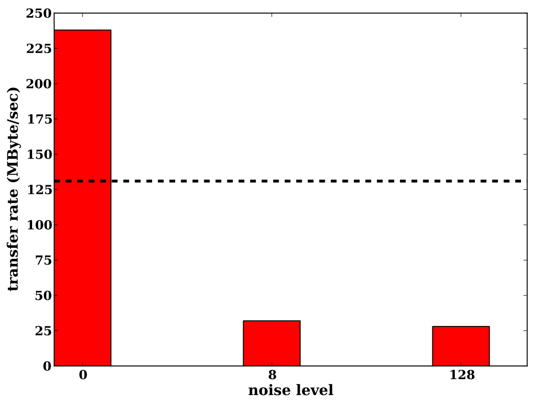

Figure 7: Benchmark using `deflate` compression with level 3. The black dashed line denotes the peak performance of the hard-drive. The noise level gives the maximum value of the random noise used to fill the frame. For 0 noise, compression works perfect and the virtual data-rate exceeds even that of the storage device. However, for non-zero noise the data-rate drops dramatically making online-compression inapplicable for high data-rate systems.

An alternative to write the HDF5 file to a RAM-disk as it is provided by tmpfs would be to use the `H5FD_CORE` driver for HDF5. Unfortunately, tests have shown that using this driver does not increase performance. One can thus conclude that although we cannot achieve raw memory performance on a RAM-disks, the performance of `tmpfs` can be reached with HDF5.

3.3 Write data with compression

As shown in section 3.1 the write-rate provided by the storage device is the limit for what can be achieved with HDF5. This limit can be virtually exceeded by compressing the data before it is written to disk. To get realistic numbers the detector frame stored in the file during the benchmark is not set to a constant value (this would be too easy) but rather set to random values simulating noise. The random numbers fluctuate between 0 and an upper limit denoted as *noise-level* here. A noise-level of 0 means no noise and a constant value of 0 will be stored to the entire detector frame. Results for the compression test are shown in Figure 7. While the first bar (no noise) in this figure clearly shows that compression can virtually increase write-rates above the physical limits of the storage device, the other bars in the plot show that in any case where the signal becomes noisy the write-rate drops far beyond the numbers one obtains for uncompressed write-operations. This leads to the following two conclusions

1. the virtual data-rate can be increased by means of compression
2. but the actual performance of compression code in HDF5 is too poor to observe this effect on realistic data.

It is thus highly recommended to tune the performance of the compression algorithms (see also section 5.2).

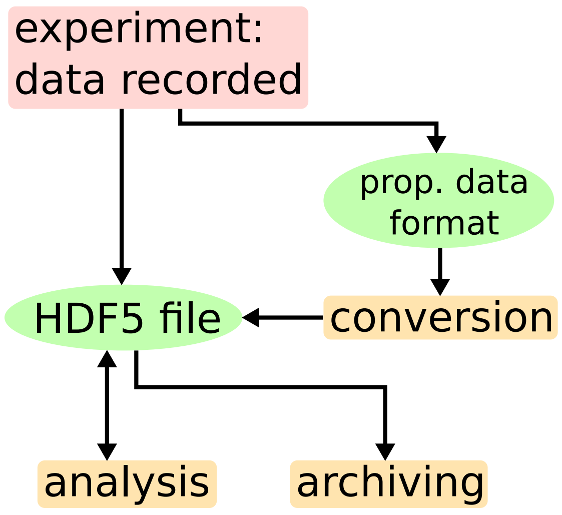

Figure 8: Data is recorded during an experiment and if not written directly to HDF5, it is converted afterwards to HDF5. The resulting HDF5 file is stored somewhere on a file server or on the local disk of a scientist. Analysis and visualization programs access data through this file. Once analysis is done the data recorded originally during the experiment must be archived. Archives are usually hybrid systems consisting of a tape library with a kind of hard-drive cache in front of it.

4 Currently planned applications

Before discussing several planed applications, a closer look on the datas' life-cycle should be taken (see Fig. 8. Data is recorded during an experiment and stored either directly as an HDF5 file or in some other facility or vendor specific data format. In the later case data will be converted to HDF5 after the experiment. The resulting HDF5 file is kept on a storage device accessible by scientists to run analysis programs on the data. Once data evaluation is finished the data must be moved to an archive for long term storage. Archives are usually tape libraries with a disk cache in front of the tapes in order to provide quick access to frequently requested data. As one can imagine such archive systems have special requirements on a data format and HDF5 basically meets these demands as will be shown later in this section.

In general two cases concerning the code involved in data storage can be derived from Fig. 8:

online-software which is code that is directly involved in recording data during the experiment. The runtime of such routines should be as deterministic as possible in order to satisfy latency demands for high data-rates.

offline-software includes all software that runs on the data after its acquisition. All kind of analysis, conversion, and archiving programs belong to this family of software. Latency is not such a big issue for this group of programs. However, reasonable performance should be provided in order to keep experiments running fluently.

Several application scenarios for both kinds of programs will be presented in the following sections of this paper.

4.1 Online data storage

Programs directly involved in data acquisition have high demands on latency and performance. This is particularly true for systems that are involved in high data-rate experiments. Figure 9 shows the basic hardware setup for such an acquisition system (for the sake of simplicity only the components which are relevant to the detector are shown). Usually the detector is controlled by a special computer which is either a Linux or an Windows PC. Concerning data storage such setups come in two flavors where

- the the control PC fetches the data from the detector hardware and stores it to disk
- or the detector has a separate direct connection to a storage device and writes data by itself (the control system tells the detector only where to store the data on its remote storage).

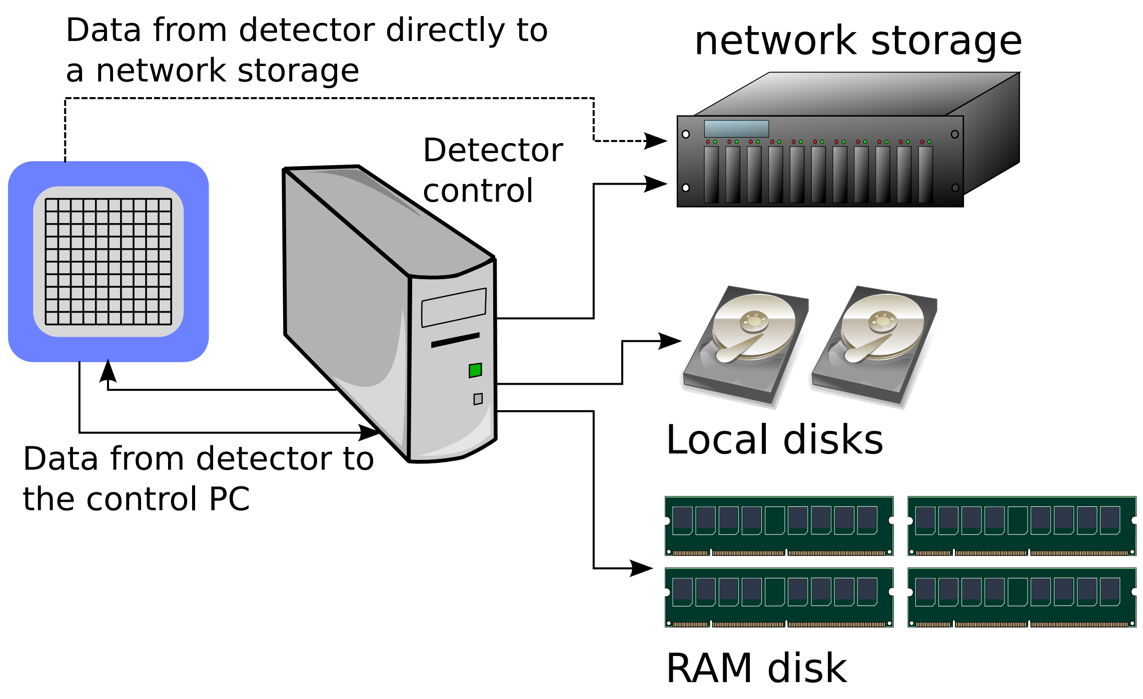
Data from detector directly to a network storage
network storage
Detector control
Local disks
Data from detector to the control PC
RAM disk

Figure 9: In this scenario the detector is controlled by a PC. There are two possibilities who detector data is stored: it an be either directly transferred from the detector to a network storage device or it can be sent to the control PC which takes care about data storage. The control PC itself can store data either on a local disk system (maybe a RAID), on a network storage device, or finally on a RAM disk.

In the later case we have hardly control over the data format the detector uses to write data. If the detector does not use HDF5 to write data a conversion program has to run after the experiment in order to pack the data into HDF5. In order to make our life easier we have to try to convince as many detector vendors as possible to support HDF5 as an output file format for their hardware. The swiss vendor DECTRIS [5] has announced that its next generation detectors will support HDF5. There are basically two criteria for a detector vendor to support a certain file format: its support by third party software and the IO performance. Since the former is not an obstacle for HDF5 it is absolutely important to improve HDF5s' performance in order to convince vendors of HDF5.

In situations where the control PC is responsible for data storage research facilities have more freedom over the choice of the file format used to store detector data. The interesting question here is: what are the requirements? To get a feeling for the data-rates involved lets have a look on some use cases. The PCO Edge camera for instance (considered as an area detector) has a resolution of 2560×2160 pixels and a dynamic range of 16 Bits. Hence, a single frame of this camera has a size of 10 MByte. The peak performance of the camera are 100 frames per second which corresponds to a data rate of about 1 GByte/sec. Another example would be a detector manufactured by Perkin Elmer. The resolution is $2\text{k} \times 2\text{k}$ with 16 Bit dynamic range for each pixel. However, scientists would like to do background correction before writing data to disk which alters the data type to 32 Bit signed integer. The resulting frame has a size of 16 MByte. Data should be recorded at a 15 frames per second which leads a data-rate of 240 MByte/sec. Although this rate can be handled by fast network storage or a RAID, the resulting data files grow large and are hard to move around. In both of the above cases compression might helps to reduce the total amount of data which needs to be stored and thus increase the virtually achievable data-rate.

It is quite clear the compression might not help in all cases. The primary reason for this is that the size of a chunk of data after compression depends heavily on the data. Thus the time required by an IO routine to transfer data is unpredictable (only an upper bound can be given which

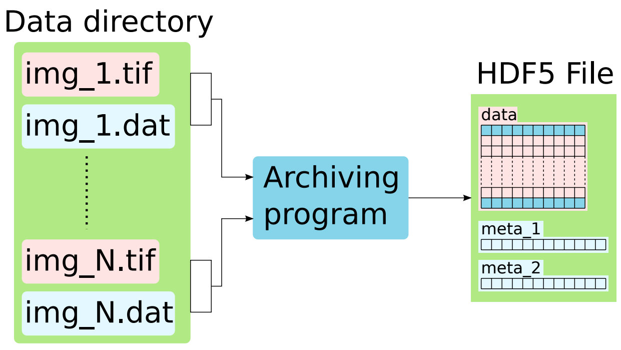

Figure 10: Here, the detector data for a single point in the measurement is stored as a single TIF file along with an ASCII file holding additional meta-data. A program gathers all this data and collates it to a single HDF5 file.

is the time required to transfer uncompressed data). This indeterministic behavior may renders compression inapplicable during the measurement. In cases where this becomes a problem data is usually stored on very fast but comparatively small storage systems without using compression. However, once the experiment is done data must be transferred to some long-term storage and this operation should not be performed within reasonable time (see below).

4.2 Offline applications - conversion, transfer, and archiving

There is a large variety of offline data processing applications like converters, archiving software, and of coarse data analysis programs. These programs act on data that is already recorded and resides on a persistent or temporary storage device. Although the requirements concerning latency are not that high for such applications as it is for online-software, performance is still an issue. In this section a couple of offline application scenarios will be presented.

4.2.1 Data transfer

Let us assume for instance an very fast experimental method where data is written at very high frame rates. Even if we would have an sufficiently fast compression algorithm at hand to deflate detector data, due to the random noise distribution in each frame the sizes of the different compressed chunks would slightly differ from each other. Consequently the time for data transfer would be slightly different for each detector frame. In order to get constant data transfer the beamline-scientist may decides to not use compression during the experiment and save data to a RAM-disk or some other other fast local storage. Once the experiment is finished the data must be transferred to some network-storage in order to release space on the temporary device. In such situations it is crucial that the transfer process runs as fast as possible since the machine cannot be used for a new experiment until the temporary storage is not empty. This would be typically an application for `h5repack` which applies a filter on one or more data-sets and creates a new file on the network-drive. However, the performance of filter codes at the moment is to low to do this within reasonable time.

4.2.2 File conversion

In many cases we have to convert data from an arbitrary format to HDF5. There are several reasons why we need to do this: one is that we might not posses control over the data format used by a detector. Another reason is legacy data from the pre-HDF5 era. In such situations a program will collect data from detector specific files and convert them to an HDF5 file as shown in Fig. 10. Collating all data into a single file has several advantages:

- only a single file descriptor must be opened for reading the data
- file-system limits concerning the maximum number of files in a directory are no longer of importance

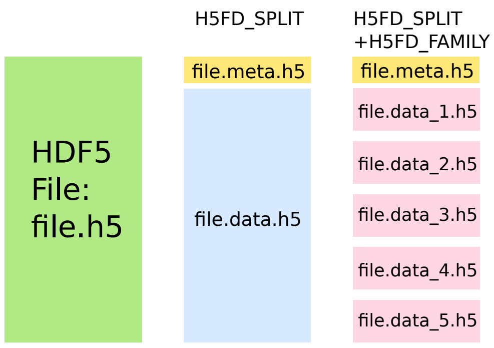

Figure 11: In order to quickly access a large HDF5 file, stored on a tape library, the file is split into a raw-data and a meta-data file where the later one can be expected to remain small. Furthermore, the large raw-data file is split into a set of smaller files making it easier to store raw-data on tapes.

- data compression in HDF5 allows reducing the total amount of space occupied by the file on disk.

The resulting single file can be easily transferred via network. Additionally compressing the data from (uncompressed) TIF files reduces the amount of time required to transfer the data over the network.

4.2.3 Archive applications

The data recorded by scientists during several campaigns must be stored in an archive. As the amount of data gathered during an experiment can grow large (from 10th to 100th of GBytes) this data is usually stored on large scale storage facilities which work with media mix consisting of hard-drives and tapes. Although being large in capacity such archive systems provide usually only very low read performance. If a single large file is written to a tape library it is split up into several parts and distributed over a bunch of tapes. If a user wants to read some data all theses tapes must be read to assemble the entire file. This is in particular bad if the user only wants to know the structure of the file to, for instance, look for an interesting data-set.

HDF5 provides an interesting approach to face this problem. A data file can be split into two parts using the `H5FD_SPLIT` driver. One of the files contains only the meta-data and can be assumed to be small. The other file holds the raw data and may grow large. The former file typically fits on a single tape and will be exclusively accessed as long as only structural information is requested by the user. As a result the access to such information is much faster than using a single HDF5 file. The result can be already observed if a large file is accessed via network storage.

There remains only one problem to solve. The file holding the raw data is still a large blob of binary data. Using the `H5FD_FAMILY` driver for this second part of the splitted HDF5 file results now in a single file holding all the meta-data and a bunch of files of constant size with the raw data. This bunch of files can be handled much easier by the tape library than a single large file.

5 Problems with the current implementation of HDF5

5.1 Archiving issues

In the previous section we have seen that HDF5 seems to be perfectly applicable for archiving data. The combination of the `H5FD_SPLIT` and `H5FD_FAMILY` drivers allows the developer to split an HDF5 file into smaller portions which are easier to handle for tape libraries. Unfortunately there remains a small problem with the files' naming convention for the two drivers. During the setup of the split-driver a file extension for the meta-data and one for the raw-data file must be passed to the function along with the file access property lists for these two files. The family driver which will be used for the raw-data file expects the file name to contain a format string which will be used to number the files appropriately. However, this format string is not used by the meta-data file. As a result you get some thing like this

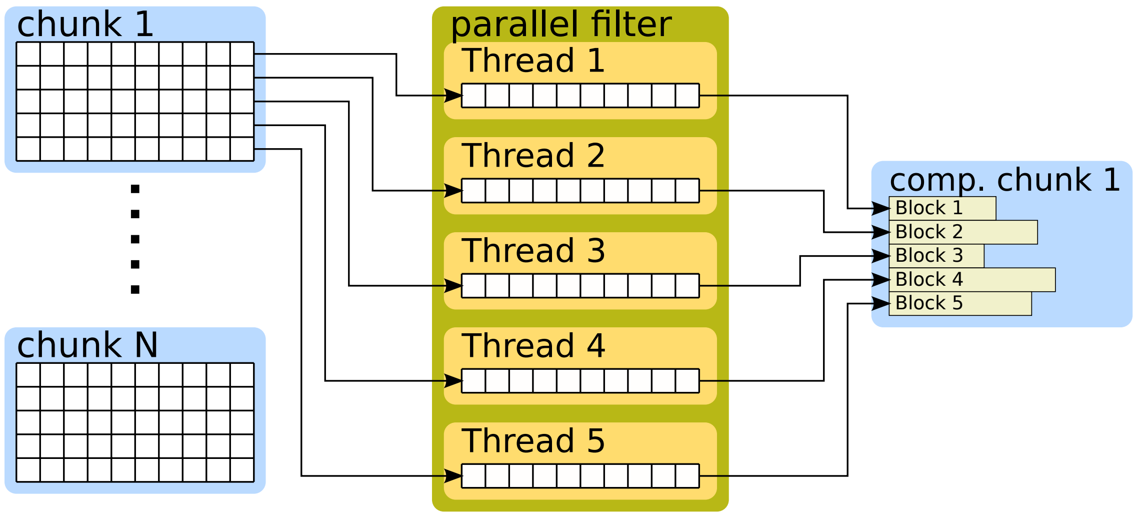

Figure 12: A possible solution to the performance problem would be to run filters in parallel on parts of the data in a single chunk. However, it must ensured to inflate the data from the compressed chunk with a single threaded filter routine in order to remain compatible with existing versions of HDF5.

```
file_%i.meta.h5
file_1.raw.h5
file_2.raw.h5
file_3.raw.h5
```

Thus it would be better to include the format string into the extension for the raw-data files so that one would obtain

```
file.meta.h5
file.raw_1.h5
file.raw_2.h5
file.raw_3.h5
```

However, this is currently not working.

5.2 Filter (compression) performance

As can be seen from the above benchmarks, HDF5 provides device performance as long it is used without compression filters (this holds even for RAM disks). Thus the bottleneck for high data-rate applications is the storage device and among those persistent storage devices like hard-disks, network drives, and RAIDs. To overcome this limitation compression of the data is required before writing the data to disk or a network storage. Unfortunately activating compression drops HDF5s' write performance to a value making it inapplicable for high performance applications. One solution to the problem would be parallelization of the IO code which can be achieved either by using MPI (and parallel MPI IO in HDF5) or multithreading. The former does not support compression of datasets. The later is doomed by the library-global mutex used within HDF5 to ensure thread-safety. Consequently all HDF5 code will execute serially even if fired from several threads. Compression is not only an important feature from the point of IO performance. The data must also be stored over a long period of time (DESY has a 10 year data storage policy for inhouse data). Due to the large amount of data recorded during XFEL or SR high data rate experiments the space data requires on disk becomes crucial. Thus compression is required in order to keep storage efforts at a minimum.

In HDF5 a chunk is the smallest amount of data to which a filter is applied and which is stored to disk. As we cannot make HDF5 fully multi-threaded it might be enough to distribute the filter code itself to several threads. In this concept, the chunks' data is split and feed into different threads for compression as depicted in Fig. 12. The compressed portions of data stored in the chunk should be inflatable by a single threaded decompression routine which is most probably the biggest issue.

5.3 The filter distribution problem

Due to the underlying physics, data recorded at SR and XFEL facilities shows some special features and properties which might can be exploited to write very efficient (application specific) filter codes. Although the actual implementation of HDF5 provides an interface to register custom filter functions for data compression, however I encounter two major problems with this approach.

At first (as far as I understood the existing HDF5 interface) the filter code must be compiled into the application using the HDF5 file. Although this is not a big deal for special purpose applications that are developed at a research facility it is nearly impossible to do this for third party applications. In particular commercial packages like Matlab or IDL would be unable to read data compressed with such an algorithm. The second problem is that I have not figured out yet how to configure custom filter code. This would be important in order to control the behavior of the compression algorithms individually for each dataset.

Thus, a generic filter interface would be highly appreciable. One can imagine a filter factory that reads a configuration file during library initialization (H5init). This configuration holds paths to shared objects (DLLs on Windows) each implementing a single filter. To add a new filter only a package with the shared object must be installed on the target system and an entry to the filter configuration file. Such an approach could be used by commercial vendors too.

Configuration of the filter routines can be done using key values pairs passed as strings to a configuration function provided by the filter code. It is up to the programmer of the filter to parse this string correctly. Using strings to encode the data circumvents the problem of handling native data types on various platforms. Finally such an approach would remove the responsibility of developing filters away from the HDF5 Group towards those who want to use a filter. The only thing that would remain to the HDF5 group is to administer filter registration.

A discussion on the HDF5 forum during October 2011 showed that such a filter interface seems to be on the wish-list of many HDF5 users.

5.4 Inserting pre-compressed data

Another interesting approach to handle the compression-performance issue was suggested by DECTRIS. In some cases detector vendors may perform data compression already on their hardware to reduce the amount of data which must be sent over the network or hardware-internal buses. It would be very efficient to store this hardware-compressed data directly to an HDF5 dataset by bypassing the filter chain in the HDF5 library. This would most probably require a hook where such data can be passed to the HDF5 code. Clearly such an approach would require custom filters to later read the data in an analysis program.

6 Conclusion

From the discussions in the previous sections on can draw the following task list to make HDF5 fully applicable for high data-rate applications:

1. fix problems when using `H5FD_FAMILY` in connection with `H5FD_SPLIT` (see section 5.1 for details)

2. implementation of parallel filters if possible (see section 5.2)

3. establish an easy to use interface for external filter code (see section 5.3)

4. provide a possibility to bypass the filter chain when writing data in order to write pre-compressed data to a file (see section 5.4).

Clearly, this list should be treated as a basement for further discussions on how to increase HDF5s' performance. Maybe there are other possibilities to achieve the goals mentioned mentioned in section 4 or to circumvent the problems described in section 5.

References

[1] ESRF, "The esrf data format." Good question - have not found useful link on the web.

[2] CBF, "Crystallographic binary format." http://www.ebi.ac.uk/pdbe/docs/exchange/mmcif/cbf/index.html.

[3] DESY, "High data rate initiative for photons, neutrons, and ions." http://www.pni-hdri.de/index_eng.html.

[4] NIAC, "Nexus - a common data format for x-ray, neutron and muon science." www.nexusformat.org.

[5] DECTRIS. http://www.dectris.com/.