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

Eugen Wintersberger

November 11, 2011

1 Motivation

Today, research facilities using synchrotrons and/or free electron lasers (FEL) as sources for x-ray radiation face 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. Since many of these formats are incapable of storing additional meta-data, which is required for data analysis, additional, mostly ASCII based, file formats where developed to store the required information. In order to access data, scientists and/or facility staff must provide code to read each of these file formats. Every time a new detector appears new code must be written or existing code adopted. Additionally one hast to take the maintenance effort into account which is required to keep these Programs up to date.

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 scale. Hence the rate at which detector data is recorded increased dramatically. High data rates have some unpleasant consequences. The first is the large number of files that will be created during an experiment. The reason for this lies in the fact that most of the existing file formats are capable of storing only a single detector image. If one takes into account that a meta-data file will eventually be written along with every detector frame one immediately gets a whole bunch of files. Such large numbers of files clearly slow down every file system which leads to bad performance of the entire system. Aside form the shear number of files also the total amount of memory consumed by the data becomes problematic. While in the early days a single experiment run consumed some MBytes of data, todays setups create hundreds of GBytes which must be stored for evaluation and later archived for documentation reasons.

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

1. selecting a common data format for all experiments
2. and developing technologies to store data using this format at high data rates.

HDF5 was finally chosen 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

[1] Which lead to a large variety of dialects unreadable to many standard readers.

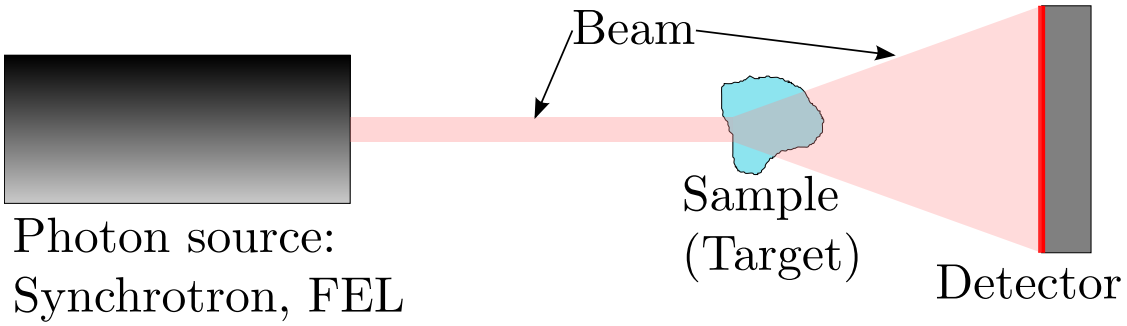

Figure 1: The basic setup for a synchrotron or free electron laser experiment. A beam of photons emitted by a source hits a sample and the scattered and/or transmitted photons are recorded by a detector.

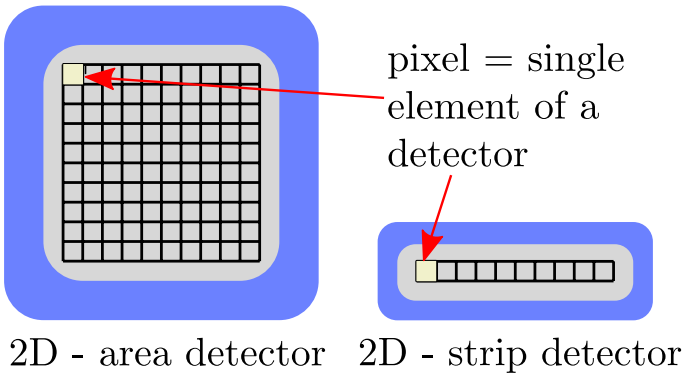

Figure 2: Two types of detectors are relevant for our HDF5 applications: 1D strip and 2D area detectors. The active (photon counting) element in each of two detector types will be referred to as *pixel* and corresponds to a single point in a multidimensional dataset.

to the various groups and data sets using NeXus [4]. Although HDF5 provides us basically with everything we need there are still problems in particular when it comes to high data rates as will be described in the ongoing sections.

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 impinge on a sample (also called target) and the transmitted and/or scattered light is recorded by a detector placed in the surrounding of the sample. Currently there are two major types of detectors 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 mentioned here. The detectors record data by counting the number of photons impinging on each of its active regions. These active (photon counting) regions of a detector (for both 1d and 2d detectors) will be referred to as *pixels*. Data is not read out continuously from a detector but after some accumulation time (exposure time) during which the detector counts photons. After this time data is fetched from the detector by the data acquisition software and saved to disk or whatever storage medium will be used. The data recorded during such a single fetch operation is also called *frame*.

This exposure and readout cycle is repeated for each data point recorded during an experiment. At each data point some parameter of the experiments' setup is varied (for instance the detector position with respect to the sample, the sample position, the temperature of the sample, etc.). After a single experiment with N points one typically ends up with N frames of detector data.

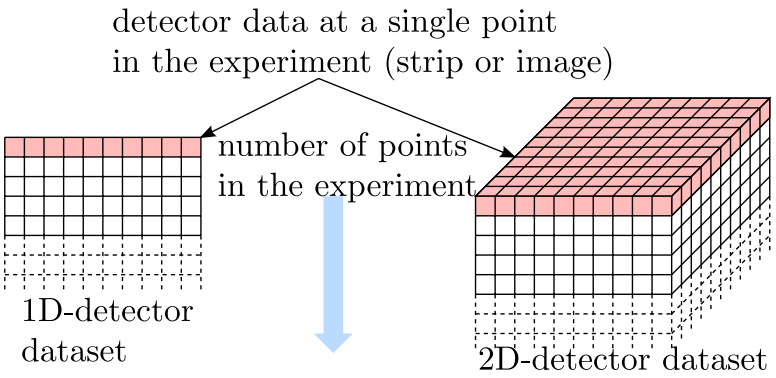

Figure 3: Organization of experiment data for 1D and 2D detectors. The first dimension of the data-set runs over the points at which data is recorded. The other dimensions correspond to the size of the detector (number of pixels along each dimension).

2.1 Data organization in the HDF5 file

Detector data should be stored in HDF5 `datasets` with the following layout. 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 (being 1 for a strip and 2 for an area detector). The maximum shape $s_{\max}$ of such a dataset 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 dataset corresponds to the number of points recorded during an experiment. Setting the maximum size of a dimension to `H5S_UNLIMITED` allows continuous expansion of the dataset along this dimension making it easy to append additional data points. In order to keep the dataset 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. The structure of a dataset for a strip and an area detector is shown in Fig. 3. 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 dataset can be considered as a container (like a list or vector in C++) holding data objects of a particular size and shape. From this point of view the first dimension of the dataset represents the container index of an element. As mentioned previously the datasets can be extended along the dimension of size `H5S_UNLIMITED`. This feature will be used this in a very excessive way: when a dataset is created its initial size along the extensible dimension will be 0 which indicates an empty container. Storing data means nothing else then appending data to such an container. Each time a new detector frame is written the dataset is extended by a single chunk. This procedure allows a system to start writing data without knowing the final number of points to write 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

We made some benchmarks using the above data layout in order to justify our choices. For benchmarks it is common to use machines tuned for this particular operation. However, since in our case the IO system must run on control PCs where other applications are active too, most of the tests where done on common PC hardware. Thus we were able to check the performance of the library under real-life conditions. The several systems where available for testing

In this section we will present only the results of the benchmarks, see appendix A for details about how we performed the tests. A short description about the hardware systems used for benchmarking is given in Tab. 1. For all test systems benchmarks are performed where data is

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.

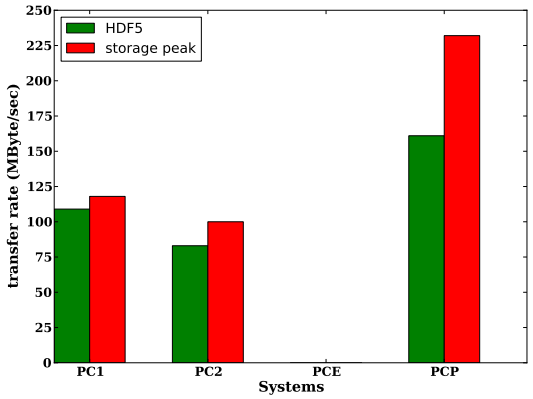

Figure 4: Comparison between HDF5 and storage device peak performance for 1024 × 1024 `unsigned short` detector frames.

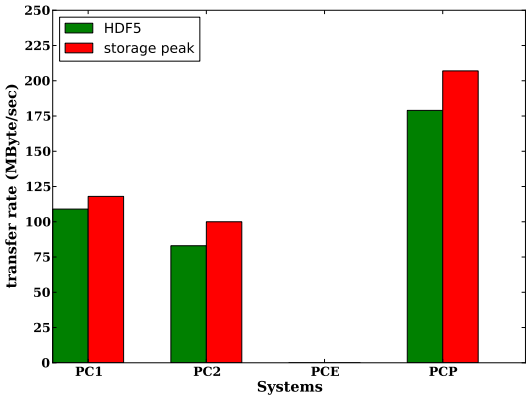

Figure 5: Comparison between HDF5 and storage device peak performance for 4096 × 4096 `unsigned short` detector frames.

written to the hard-drive setup in the machine and a RAM disk.

The code for the benchmarks is quite straight forward, no special tweaks were made to HDF5 data structures. As sample data for the tests we assumed a 2D area detector so the resulting data-set is a 3D block as depicted in Fig. 3. The data is assumed to be of type `unsigned short`.

3.1 Write to disk, RAID, and network storage

The first benchmark presented considers write-operation to conventional persistent storage devices including hard-disks, RAID systems, and network storage. 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. The size of the detector frame is only important for network storage as can be seen from a comparison of the two PCP columns in Figures 4 and 5. This is a quite important information. The question raises if one could not circumvent this effect by tuning the underlying network file-system to obtain optimum performance already at the smallest frame size available by the detectors provided at the beamline or facility.

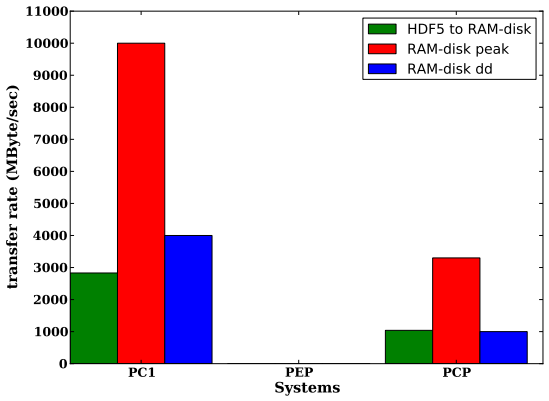

Figure 6: Benchmark on RAM-disks on various testing systems. Here the write performance drops significantly below the devices' capabilities. One reason might be that the data has to be copied from one location in memory to another. However, the HDF5 performance is close to what we can achieve with dd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

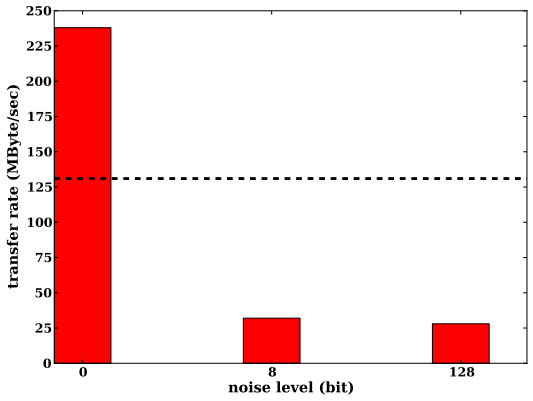

Figure 7: Benchmark using `deflate` compression with level 3. The black dashed line denotes the peak performance of the hard-drive. The noise level gives the maximum value of the random noise used to fill the frame. For 0 noise compression works perfect and the virtual data-rate exceeds even that of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

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 good temporary storage in cases where measurements have to run extremely fast. 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. This benchmarks measures raw RAM performance. However, we are more interested in the performance of the `tmpfs` partition onto which the data was dumped. This was measured using `dd` and the results are depicted by the blue bars in Fig. 6. If we compare the HDF5-data rate with the one obtained from `dd` we are back at device performance.

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. However, 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.

3.3 Write data with compression

As shown in the previous section **??** the device performance is the ultimate limit for write-performance. However, one could increase this number virtually by compressing the data before writing as it is possible with HDF5. 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 to simulate noise. However, the noise has an upper limit given by the *noise-level*. A noise-level of 0 would mean no noise and a constant value of 0 will be stored to the entire detector frame. Figure 7 shows the results for the compression test. While the first bar (no noise) in this

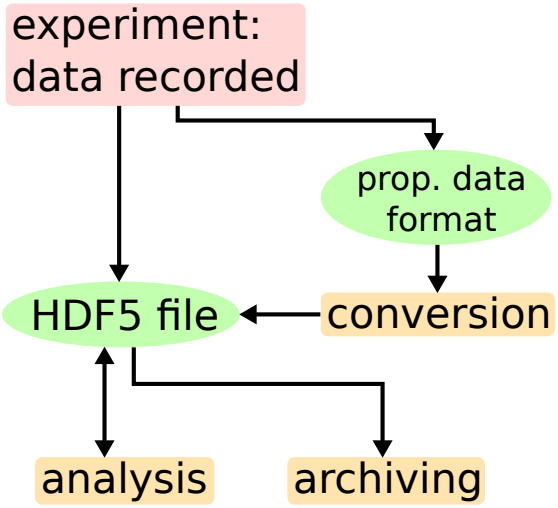

Figure 8: The data is recorded during an experiment and if not written directly to HDF5, it is converted afterwards to HDF5. The resulting HDF5 file is stored somewhere on a file server or even on the local disk of a scientist. Analysis and visualization programs access data through this file. Once analysis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

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).

4 Currently planned applications

Before talking about concrete applications one should have a look on the planed life-cycle of data as depicted in 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. The resulting HDF5 file is kept on a quick 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 an HDF5 can meet these demands as will be shown later in this section.

In general two cases concerning the code involved in data strorage 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 which is basically all code that runs on the data after is 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 applications scenarios for both types of applications will be presented in the following sections of this paper.

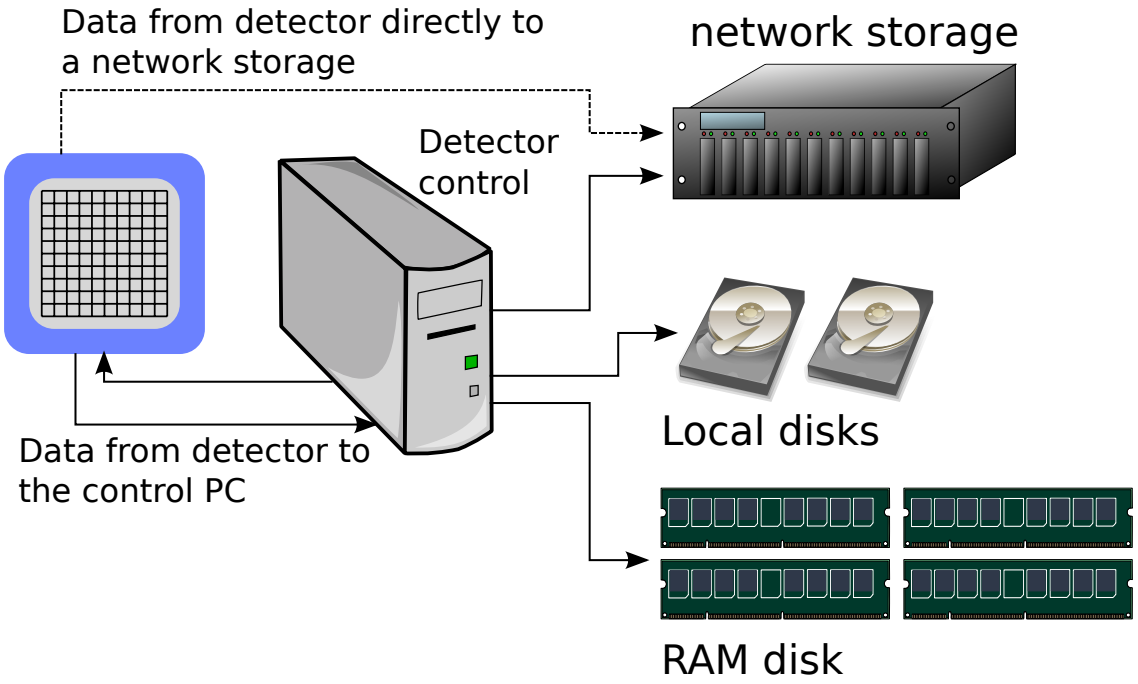

Figure 9: In this scenario the detector is controlled by a PC. There are two possibilities who detector data is stored: it an be either directly transfered from the detector to a network storage device or it can be sent to the control PC which takes care about data storage. The control PC itself can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

4.1 Online data storage

Programs directly involved in data acquisition have high demands on latency and performance. This is in particular 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 for the detector, which is the major data source, 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).

In the later case we have hardly control over the data format the detector uses to write data. In cases where the detector is not using HDF5 directly to store data a conversion program has to run after the experiment in order to pack the detector data into HDF5. It is thus highly important to convince as many vendors of such detectors to directly 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. For detector companies IO performance is a critical issue. Therefore, increasing HDF5s' performance is critical to convince them to use HDF5 for their applications.

In situations where the control PC is responsible for data storage research facilities have more freedom of choice over the data format used. The interesting question here is: what are the requirements? To get a feeling for the data-rates involved lets have a look some applications. The PCO Edge camera for instance (considered as an area detector) has a resolution of 2560×2160 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. From the benchmark section we have seen that directly using HDF5 is only possible by using a RAM-disk. Another example would be a detector manufactured by Perkin Elmer. The resolution is 2k × 2k 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 gives 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 increases the virtually achievable data rate.

It is quite clear the compression might not help in all the above 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 with 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 (like a RAM-disk) without using compression. However, once the experiment is done data must be transfered 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. In all these cases the data is already recorded and resides on a persistent or temporary storage device (like a RAM-disk). Although the requirements concerning latency are not that high for such applications as in comparison to code that runs during a measurement, 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. As a consequence the time for data transfer would be slightly different for each detector frame. In order to get constant data transfer the beamline-scientist may has decided to not use compression during the experiment and use a RAM-disk or some other other fast local storage. Once the experiment is finished the data must be transfered to some network-storage in order to release space on the temporary storage. 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 datasets 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 do not posses control over the data format used by a detector. Another reason is maybe 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

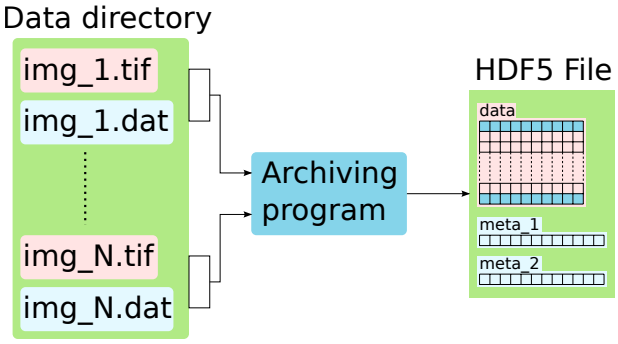

Figure 10: Here, the detector data for a single point in the measurement is stored as a single TIF file along with an ASCII file holding additional meta-data. A program gathers all this data and collates it to a single HDF5 file.

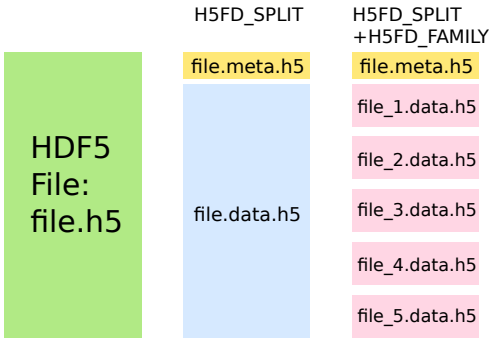

Figure 11: In order to make accessing a large HDF5 file, stored on a tape library, the file is split into a raw-data and a meta-data file where the later one can be expected to remain small. The large raw-data file is split into a set of smaller files making it easier to store raw-data on tapes.

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

The resulting single file can be easily transfered 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 (usually LTO). As a result all data access runs via network which is naturally much slower than local storage. In particular tape libraries dramatically decrease 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. Using the SPLIT virtual file driver results in tow files where one holds only the meta-data and the second the raw data. The former one is typically small and fits to a single tape. Now, if a user only requests structural information from a file only this small meta-data file will be touched. As a result the access to such information is much faster than using a single HDF5 file.

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 split 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 naming scheme 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

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

instead of

```
file.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

Currently HDF5 is the only data format that supports all features required to save data from SR and XFEL experiments along with meta-data. The library is sufficiently effective and yields device performance, as shown in the previous section, if used without any filters (compression, checksum). This holds even if the data is written to a RAM disk. Thus the bottleneck for HDR 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 if compression is activated for a dataset the write performance drops dramatically making HDF5 inapplicable for HDR 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. 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

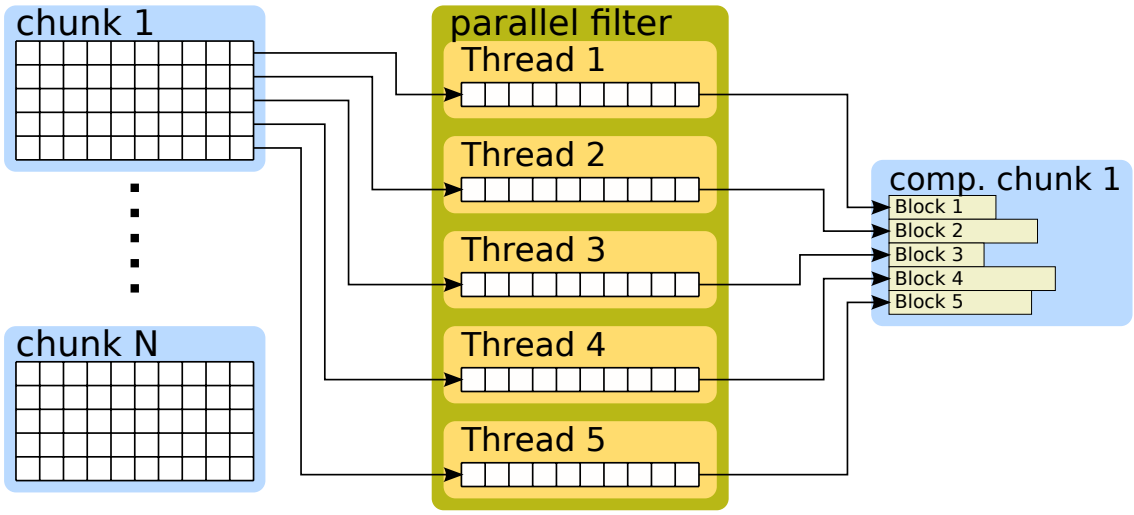

Figure 12: A possible solution to the performance problem would be to run filters in parallel on parts of the data in a single chunk. However, it must ensured to inflate the data from the compressed chunk with a single threaded filter routine in order to remain compatible with existing versions of HDF5.

code itself to several threads. In this concept, the chunks' data is split and feed into different threads for compression. The concept is depicted in Fig. 12 for the case of a compression filter. The compressed portions of data stored in the chunk should be inflatable by a single threaded decompression routine. This is most probably one of the biggest issues for the entire procedure.

Exploiting certain properties of data recorded at SR and XFEL experiments special filters may can further improve the performance. Using custom filters, however, introduces another problem - see the next section for this 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. These circumstance might be exploited in writing special compression codes in order to get chunk compression done very efficiently. Although the actual implementation of HDF5 provides an interface to register custom filter functions for data compression, this approach has several difficulties.

The first problem is that the code of the filter function must be incorporated in the program that wants to use it. This means usually to recompile the program. Although this is not a big deal for special purpose applications that are developed at a research facility it is nearly impossible to do so for third party applications. In particular commercial packages like Matlab or IDL would be unable to read data compressed with such an algorithm. In addition I have not figured out yet how to pass additional parameters. This would be important in order to control the behavior of the compression algorithms individually for each dataset.

A generic filter interface would be highly appreciable. One can imagine a filter factory that reads through a configuration file during the initialization of HDF5 (H5init). This configuration holds paths to shared objects (DLLs on Windows) each providing a certain 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 facility. It is up to the programmer of the filter code to parse these key value pairs 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.

type size (Bytes)	frame shape (nx,ny)	block size (MBytes)	counts	total (MBytes)
2	(1024, 1024)	2	10000	20000
2	(2048, 2048)	8	2500	20000
2	(4096, 4096)	32	625	20000

Table 2: Blocksize and count parameters for `dd` to test native harddrive and RAID performance. The resulting file should have a size of 20 GByte.

A How benchmarks were performed

A.1 Compiling HDF5 1.8.7

To obtain optimum results we used not prepackaged editions of HDF5 provided by the Linux distributors but rather compiled the actual version from the plain vanilla sources by ourself. We used HDF5 version 1.8.7 for the benchmarks shown in this paper. The library was compiled only with C-support including thread-safety using the following configuration:

```
$> ./configure --prefix=/home/eugen/Applications/inst --disable-fortran \
 --disable-cxx --with-pthread=/usr --enable-threadsafe
$> make -j4
$> make
$> make install
```

The compiler used was `gcc` version 4.6.1 shipped with Ubuntu 11.10 on an AMD64 system.

A.2 Testing device performance

In order to have a reference data rate the *native* performance of each storage device was measured. For harddrives and RAID systems a simple `dd` was used on a terminal. We used 16 Bit unsigned integer as the frames' data and set the block size to the size of a single detector frame. The count parameters was adjusted so that the resulting output file has a size of 20 GByte.

```
$> dd if=/dev/zero of=bin.dat bs=<frame size> count=<number of frames>
```

For RAM disks the native RAM performance was measured using the `STREAM` benchmark [6].

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/.

[6] J. D. McCalpin, "Stream: Sustainable memory bandwidth in high performance computers." http://www.cs.virginia.edu/stream/.