HDF5 Compression Demystified

Why it sometimes doesn't seem to work?

Elena Pourmal

The purpose of this document is to help the HDF5 users with troubleshooting problems with HDF5 filters, especially with compression filters. The document assumes that the reader knows HDF5 basics, is aware of the compression feature in HDF5 and is seeking information on how to use it effectively. We refer the reader who is not familiar with the feature to the HDF5 Introductory Tutorial [1], and the other documentation cited in the Reference section of this document.

Table of Contents

Introduction

The HDF5 library allows data to pass through a user-defined filter pipeline in order to modify them during I/O operations. Modification methods (or filters) provided by the HDF5 library, also called predefined filters, include several types of data compression, data shuffling and checksum generation [2]. Users can implement their own filters and use them with the HDF5 library [3, 4]. While the programming model and usage of the filters are straightforward, it is easy, especially for a novice user, to overlook certain details, including the HDF5 filters properties, and end up with data in an HDF5 file that is not compressed as they would expect.

This document provides an overview of why compression may not work and presents a few troubleshooting techniques to help diagnose the underlying issues.

Summary: Why HDF5 compression sometimes doesn't work

This section provides a short overview of the problem and explains the structure of the rest of the document.

Sometimes a user finds that HDF5 data was not compressed in the file or that the compression ratio is very small. This fact itself doesn't mean that compression "didn't work". It rather suggests that something might have gone wrong when a compression filter was applied. How can one find out?

There are two major reasons why a filter didn't produce the desired result:

1. The filter was not applied.
2. The filter was applied but was not effective.

If the filter wasn't applied at all, there is only one reason: it was not included at compile time when the library was built or not found at run time for dynamically loaded filters. The absence or presence of HDF5 predefined filters can be confirmed by examining installed HDF5 files or with the help of HDF5 API calls (section 3). The absence or presence of all filter types can be confirmed by running HDF5 command-line utilities, such as `h5dump` with the `–pH` flags, on the produced HDF5 files (section 4).

In particular, section 3.1 talks about configuring the HDF5 library with external and internal filters and how this configuration may go wrong. The material in section 3.2 explains the HDF5 library's behavior in the absence of filters. Section 3.3 shows several ways to establish if a predefined filter is configured in, by examining the installed HDF5 software including header files and the library binary file. It also shows how applications can check the availability of any filter at run time and how to avoid creating files without a filter being applied.

If the library, which was used to create a file, is not available to a user, he/she can use HDF5 command line tools such as `h5dump`, `h5ls` and `h5debug` to find out if a filter was applied. See section 4 for details.

Effectiveness of compression filters is a complex matter and is not a subject of this document. Section 5 gives a short overview of the problem and provides an example in which the advantages of different compression filters and their combinations are shown.

Appendix section of the document contains the sample code used to create one of the files discussed in section 4.

How to detect that a compression filter was not configured in

This section examines how it may happen that a filter, including a compression filter, is not available to an application and describes the behavior of the HDF5 library in the absence of the filter. Then we discuss how to troubleshoot the problem by checking the HDF5 installation and what an application can do at run time to check if a filter is available.

How the HDF5 library configuration may miss a compression filter

The HDF5 library uses external libraries for data compression. There are two predefined compression methods, GZIP (deflate) and SZIP, which can be requested at the HDF5 library configuration time (compile time). User-defined compression filters and the corresponding libraries are usually linked with an application, or provided as a dynamically loaded library [3, 4].

GZIP (deflate) and SZIP require the libz.a(so) and libsz.a(so) libraries, respectively, to be present on the system and to be enabled during HDF5 configuration by with this configure command:

```
./configure –with-zlib=/path… –with-szlib=/path… <other flags>
```

There is one important difference in the behavior of configure between GZIP(deflate) and SZIP.

On UNIX systems, *when GNU autotools are used to build HDF5*, the GZIP(deflate) compression is enabled *automatically*, if the zlib library is present on the system in default locations, without explicitely specifying –with-zlib=/path… For example, if libz.so is installed under /usr/lib with the header under /usr/include or under /usr/local/lib with the header under /usr/local/include, the following HDF5 configure command will find the GZIP library and will configure the compression filter in.

```
./configure
```

The SZIP compression should *always* be *requested* with the configure flag. Configure will not fail if libraries supporting the requested compression method are not found, for example, because a specified path was not correct, or the library is missing.

When the library is built with CMake, GZIP(deflate) and SZIP compression filters are enabled by default in the source code distribution's config/cmake/cacheinit.cmake file. See the CMake installation instructions for locations of the libraries.

If compression is not requested or found at configuration time, the compression method is not registered with the library and cannot be applied when data is written or read. For example, the h5repack tool will not be able to remove an szip compression filter from a dataset, if the szip library was not configured into the library against which the tool was built. The next section discusses the behavior of the HDF5 library in the absence of filters.

Note that there are four internal predefined filters - nbit, fletcher32, scaleoffset and nbit. They are enabled by default by both configure and CMake builds. While they can be disabled intentionally with a configure flag –disable-filters, in practice it is a very rare event.

Therefore, the discussion and the examples in this document will focus mainly on compression filters, but everything said applies to other missing internal filters as well.

How does the HDF5 library behave in the absence of a filter

By design, the HDF5 library allows applications to create and write datasets using filters that are not available at creation/write time. This feature makes it possible to create HDF5 files on one system and to write data on another system where the HDF5 library is configured with or without the requested filter.

Let' s recall the HDF5 programming model for enabling filters.

An HDF5 application uses one or more H5Pset_<filter> calls to configure a dataset's filter pipeline at its creation time. The excerpt below shows how a deflate filter is added to a pipeline.

```
/*
 * Create the dataset creation property list, add the gzip
 * compression filter and set the chunk size.
 */
dcpl = H5Pcreate (H5P_DATASET_CREATE);
status = H5Pset_deflate (dcpl, 9);
status = H5Pset_chunk (dcpl, 2, chunk);
dset = H5Dcreate (file, DATASET,…, dcpl,…);
```

For all internal filters (nbit, fletcher32, scaleoffset and nbit) and the external deflate filter, the HDF5 library does **not** check if the filter is registered when the corresponding H5Pset_<filter> is called. The only exception to this rule is H5Pset_szip, which will fail if szip was not configured in or is configured with a decoder only. Hence, in the example above H5Pset_deflate will succeed. The specified filter will be added to the dataset's filter pipeline and will be applied to any data written to this dataset.

When H5Pset_<filter> is called, a record for the filter is added to the dataset's object header in the file and information about the filter can be queried with the HDF5 APIs and displayed by HDF5 tools such as h5dump. The presence of filter information in a dataset's header doesn't mean that the filter was actually applied to the dataset's data, as will be explained later in this document. Section 4 shows how to use h5ls and h5debug tools to determine if the filter was actually applied.

The success of further write operations to a dataset when filters are missing depends on the filter type.

By design, an HDF5 filter can be optional or required. This filter mode defines the behavior of the HDF5 library during write operations. In the absence of an optional filter, H5Dwrite calls will succeed and data will be written to the file, bypassing the filter. A missing required filter will cause H5Dwrite calls to fail. Clearly, H5Dread calls will fail when filters that are needed to decode the data are missing.

The HDF5 library has only one required internal filter, Fletcher32 (checksum creation), and one required external filter, SZIP. As mentioned earlier, only the SZIP compression (H5Pset_szip)

will flag the absence of the filter. If, despite the missing filter, an application goes on to create a dataset via `H5Dcreate`, the call will succeed, but the `SZIP` filter will **not** be added to the filter pipeline. This behavior is different from all other filters that may not be present, but will be added to the filter pipeline and applied during I/O. Please see the discussion in section 3.3.5 on how to determine if a filter is available and to avoid writing data while the filter is missing.

Developers, who create their own filters, use the "`flags`" parameter in `H5Pset_filter` to declare if the filter is optional or required. One can determine the filter type by calling `H5Pget_filter` and checking the value of the "`flags`" parameter.

For more information on filter behavior in HDF5 see [3].

How to determine if the HDF5 library was configured with a compression filter needed

The previous section described how the HDF5 library could be configured without certain compression filters and the expected library behavior.

The following subsections explain how to determine if a compression method is configured in the HDF5 library and how to avoid accessing data if the filter is missing.

Examine the hdf5lib.settings file

In order to check how the library was configured and build, one should examine the `hdf5lib.settings` text file found in the `lib` directory of the HDF5 installation point and search for the lines that contain the "`I/O filters`" string. The `hdf5lib.settings` file is automatically generated at configuration time when the HDF5 library is built, with configure on UNIX or with `CMake` on UNIX and Windows, and it should contain the following lines:

```
I/O filters (external): deflate(zlib),szip(encoder)
I/O filters (internal): shuffle,fletcher32,nbit,scaleoffset
```

These lines show the compression libraries configured with HDF5. Here is an example of the same output when external compression filters are absent:

```
I/O filters (external):
I/O filters (internal): shuffle,fletcher32,nbit,scaleoffset
```

Please notice that the same lines in the file generated by `CMake` look slightly different:

```
I/O filters (external): DEFLATE ENCODE DECODE
I/O filters (internal): SHUFFLE FLETCHER32 NBIT SCALEOFFSET
```

"`ENCODE DECODE`" indicates that both `SZIP` compression encoder and decoder are present. This inconsistency between configure and `CMake` generated files will be removed in a future release.

If the `hdf5lib.settings` file is not present on the system, then one can examine a public header file or the library binary file to find out if a filter is present, as is discussed in the next two sections.

Examine the H5pubconf.h header file

In order to find if a filter is present one can also inspect the HDF5 public header file

installed under the `include` directory of the HDF5 installation point. If compression and internal filters are present, the corresponding symbols will be defined as follows:

```
/* Define if support for deflate (zlib) filter is enabled */
#define H5_HAVE_FILTER_DEFLATE 1

/* Define if support for Fletcher32 checksum is enabled */
#define H5_HAVE_FILTER_FLETCHER32 1

/* Define if support for nbit filter is enabled */
#define H5_HAVE_FILTER_NBIT 1

/* Define if support for scaleoffset filter is enabled */
#define H5_HAVE_FILTER_SCALEOFFSET 1

/* Define if support for shuffle filter is enabled */
#define H5_HAVE_FILTER_SHUFFLE 1

/* Define if support for szip filter is enabled */
#define H5_HAVE_FILTER_SZIP 1
```

If a compression or internal filter was not configured, the corresponding lines will be commented out as follows:

```
/* Define if support for deflate (zlib) filter is enabled */
/* #undef H5_HAVE_FILTER_DEFLATE */
```

Check the HDF5 library's binary

The HDF5 library's binary contains summary output similar to what is stored in the `hdf5lib.settings` file. One can use the UNIX "`strings`" command to get information about the configured filters:

```
% strings libhdf5.a(so) | grep "I/O filters ("
      I/O filters (external): deflate(zlib),szip(encoder)
      I/O filters (internal): shuffle,fletcher32,nbit,scaleoffset
```

When compression filters are not configured, the output of the command above will be:

```
      I/O filters (external):
      I/O filters (internal): shuffle,fletcher32,nbit,scaleoffset
```

On Windows one can use the `dumpbin /all` command and then view and search the output for strings like "`DEFLATE`", "`FLETCHER32`", "`DECODE`", "`ENCODE`", etc.

.....

```
  10201860: 4E 0A 20 20 20 20 20 20 20 20 20 49 2F 4F 20 66  N.         I/O f
  10201870: 69 6C 74 65 72 73 20 28 65 78 74 65 72 6E 61 6C  ilters (external
  10201880: 29 3A 20 20 44 45 46 4C 41 54 45 20 44 45 43 4F  ):  DEFLATE DECO
  10201890: 44 45 20 45 4E 43 4F 44 45 0A 20 20 20 20 20 20  DE ENCODE.
  102018A0: 20 20 20 49 2F 4F 20 66 69 6C 74 65 72 73 20 28     I/O filters (
  102018B0: 69 6E 74 65 72 6E 61 6C 29 3A 20 20 53 48 55 46  internal):  SHUF
```

```
  102018C0: 46 4C 45 20 46 4C 45 54 43 48 45 52 33 32 20 4E  FLE FLETCHER32 N
  102018D0: 42 49 54 20 53 43 41 4C 45 4F 46 46 53 45 54 0A  BIT SCALEOFFSET.
…..
```

Check the compiler script

One can also use the compiler scripts, for example, `h5cc` to verify that a compression library is present and configured in. Use the "- `show`" option with any of the compilers scripts found in the bin subdirectory of the HDF5 installation directory. If found, one will see the `–lsz` and `–lz` options among the linker flags:

```
$ h5cc -show
gcc -D_LARGEFILE_SOURCE -D_LARGEFILE64_SOURCE -D_BSD_SOURCE -
L/mnt/hdf/packages/hdf5/v1812/Linux64_2.6/standard/lib
/mnt/hdf/packages/hdf5/v1812/Linux64_2.6/standard/lib/libhdf5_hl.a
/mnt/hdf/packages/hdf5/v1812/Linux64_2.6/standard/lib/libhdf5.a -lsz -lz -lrt -ldl
-lm -Wl,-rpath -Wl,/mnt/hdf/packages/hdf5/v1812/Linux64_2.6/standard/lib
```

Using HDF5 APIs

Applications can check filter availability at run time. In order to check the filter's availability with the HDF5 library one has to know the filter ID number (e.g., 1 for `deflate`) or a corresponding symbol (e.g., `H5Z_FILTER_DEFLATE`) and call the `H5Zfilter_avail` function as shown in the example below. Use `H5Zget_filter_info` to determine if the filter is configured to decode data, to encode data, neither, or both.

```
    /*
     * Check if gzip compression is available and can be used for both
     * compression and decompression.
     */
    avail = H5Zfilter_avail(H5Z_FILTER_DEFLATE);
    if (!avail) {
        printf ("gzip filter not available.\n");
        return 1;
    }
    status = H5Zget_filter_info (H5Z_FILTER_DEFLATE, &filter_info);
    if ( !(filter_info & H5Z_FILTER_CONFIG_ENCODE_ENABLED) ||
                !(filter_info & H5Z_FILTER_CONFIG_DECODE_ENABLED) ) {
        printf ("gzip filter not available for encoding and decoding.\n");
        return 1;
    }
```

`H5Zfilter_avail` can be used to find filters that are registered with the library or are available via dynamically loaded libraries, see [4].

Currently there is no HDF5 API call to retrieve a list of all registered filters or dynamically loaded filters. The default installation directory for HDF5 dynamically loaded filters are `"/usr/local/hdf5/lib/plugin"` on UNIX and `"%ALLUSERSPROFILE%\hdf5\lib\plugin"` on Windows. One can also check if the environment variable `HDF5_PLUGIN_PATH` is set on the system and refers to a directory with available plugins.

1)

How to use HDF5 tools to investigate missing compression filters

In this section we will use the HDF5 command line utilities to examine if a file was created with an HDF5 library that did or didn't have a compression filter configured in.

How to use h5dump to examine files with compressed data

One should use the `–p` flag to display dataset properties including compression filters and the `–H` flag to suppress output of the data. The program provided in the Appendix of this document creates a file called `h5ex_d_gzip.h5`. The output of `h5dump` shows that the deflate compression filter was added to the `DS1` dataset filter pipeline at creation time.

```
$ hdf5/bin/h5dump -p -H *.h5
HDF5 "h5ex_d_gzip.h5" {
GROUP "/" {
   DATASET "DS1" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 32, 64 ) / ( 32, 64 ) }
      STORAGE_LAYOUT {
         CHUNKED ( 5, 9 )
         SIZE 5018 (1.633:1 COMPRESSION)
      }
      FILTERS {
         COMPRESSION DEFLATE { LEVEL 9 }
      }
      FILLVALUE {
         FILL_TIME H5D_FILL_TIME_IFSET
         VALUE  0
      }
      ALLOCATION_TIME {
         H5D_ALLOC_TIME_INCR
      }
   }
}
}
```

The output also shows a compression ratio, which is defined as (original size)/(storage size). The size of the stored data is 5018 bytes vs. 8192 bytes of uncompressed data, a ratio of 1.663. Clearly the filter was successfully applied.

Now let's look at what happens when the same program is linked against an HDF5 library that was not configured with the `GZIP` library.

Notice that some chunks are only partially filled. 56 chunks (7 along the first dimension and 8 along the second dimension) are required to store the data. Since no compression was applied, each chunk has size 5x9x4 = 180 bytes, resulting in a total storage size of 10,080 bytes. With an original size of 8192 bytes, the compression ratio is 0.813 (i.e., less than 1!) and visible in the output below.

```
$ hdf5/bin/h5dump -p -H *.h5
HDF5 "h5ex_d_gzip.h5" {
GROUP "/" {
   DATASET "DS1" {
      DATATYPE  H5T_STD_I32LE
      DATASPACE  SIMPLE { ( 32, 64 ) / ( 32, 64 ) }
      STORAGE_LAYOUT {
         CHUNKED ( 5, 9 )
         SIZE 10080 (0.813:1 COMPRESSION)
      }
      FILTERS {
         COMPRESSION DEFLATE { LEVEL 9 }
      }
      FILLVALUE {
         FILL_TIME H5D_FILL_TIME_IFSET
         VALUE  0
      }
      ALLOCATION_TIME {
         H5D_ALLOC_TIME_INCR
      }
   }
}
}
```

As discussed in section 3.2, the presence of a filter in an object's filter pipeline **doesn't imply** that it will be applied unconditionally when data is written.

If the compression ratio is less than 1, compression was not applied. If it is 1, and compression is shown by `h5dump`, more investigation is needed and discussed in the next section.

How to use h5ls and h5debug to find a missing compression filter

Filters operate on chunked datasets. A filter may be ineffective for one chunk (e.g., the compressed data is bigger than the original data), and succeed on another. How can one discern if a filter is missing or just ineffective (and as a result non-compressed data was written)? Use `h5ls` and `h5debug` to investigate the issue!

First, let's take a look at what kind of information `h5ls` displays about the dataset `DS1` in our example file, which was written with an HDF5 library that has the `deflate` filter configured in:

```
$ h5ls -vr h5ex_d_gzip.h5
Opened "h5ex_d_gzip.h5" with sec2 driver.
/                        Group
    Location:  1:96
    Links:     1
/DS1                     Dataset {32/32, 64/64}
    Location:  1:800
    Links:     1
    Chunks:    {5, 9} 180 bytes
    Storage:   8192 logical bytes, 5018 allocated bytes, 163.25% utilization
    Filter-0:  deflate-1 OPT {9}
    Type:      native int
```

We see output similar to `h5dump` output with the compression ratio at 163%.

Now let's compare this output with another dataset `DS1, this time` written with a program linked against an HDF5 library without the `deflate` filter present.

```
$ h5ls -vr h5ex_d_gzip.h5
Opened "h5ex_d_gzip.h5" with sec2 driver.
/                        Group
    Location:  1:96
    Links:     1
/DS1                     Dataset {32/32, 64/64}
    Location:  1:800
    Links:     1
    Chunks:    {5, 9} 180 bytes
    Storage:   8192 logical bytes, 10080 allocated bytes, 81.27% utilization
    Filter-0:  deflate-1 OPT {9}
```

```
    Type:      native int
```

The h5ls tool shows that the deflate filter was added to the filter pipeline of the dataset DS1. It also shows that the compression ratio is less than 1. We can confirm by using h5debug that the filter was not applied at all, and, as a result of the missing filter, the individual chunks were not compressed.

From the h5ls output we know that the dataset object header is located at address 800. We retrieve the dataset object header at address 800 and search the layout message for the address of the chunk index B-tree as shown in the excerpt of the h5debug output below:

```
$ h5debug h5ex_d_gzip.h5 800
Reading signature at address 800 (rel)
Object Header...
…..
Message 4...
   Message ID (sequence number):                0x0008 `layout' (0)
   Dirty:                                       FALSE
   Message flags:                               <C>
   Chunk number:                                0
   Raw message data (offset, size) in chunk:    (144, 24) bytes
   Message Information:
      Version:                                  3
      Type:                                     Chunked
      Number of dimensions:                     3
      Size:                                     {5, 9, 4}
      Index Type:                               v1 B-tree
      B-tree address:                           1400
```

Now we can retrieve the B-tree information:

```
$ h5debug h5ex_d_gzip.h5 1400 3
Reading signature at address 1400 (rel)
Tree type ID:                                   H5B_CHUNK_ID
Size of node:                                   2616
Size of raw (disk) key:                         32
Dirty flag:                                     False
Level:                                          0
```

```
Address of left sibling:                    UNDEF
Address of right sibling:                   UNDEF
Number of children (max):                   56 (64)
Child 0...
   Address:                                 4016
   Left Key:
      Chunk size:                           180 bytes
      Filter mask:                          0x00000001
      Logical offset:                       {0, 0, 0}
   Right Key:
      Chunk size:                           180 bytes
      Filter mask:                          0x00000001
      Logical offset:                       {0, 9, 0}
Child 1...
   Address:                                 4196
   Left Key:
      Chunk size:                           180 bytes
```

We see that the size of each chunk is 180 bytes, i.e., compression was not successful. The filter mask value `0x00000001` indicates that filter was not applied; see [6] for more information on how to interpret the filter mask.

What to do if a compression filter was not effective

There are several things one can do to improve effectiveness of compression in HDF5:

1. Find the compression that is "right" for the type of data and for the objectives one tries to achieve. For example, SZIP compression is fast but typically achieves smaller compression ratios for floating point data than GZIP, as was shown in [7]. Which one is the better fit? The answer will be different for different applications, data providers and data consumers.
2. Once you have the right compression method, find the right parameters. For example, deflate compression at level 6 usually achieves a compression ratio comparable to level 9, in less time.
3. As will be shown in section 5.1, data preprocessing (e.g., shuffling [2]) in combination with compression can drastically improve the compression ratio.
4. Look beyond compression. An HDF5 file may contain a substantial amount of unused space. See discussion in section 5.2.

The `h5repack` tool can be used to experiment with the data to address items 1 – 3. In order to address item 4, use the `h5stat` tool to determine if space is used efficiently in an HDF5 file. The `h5repack` tool can be used to reduce the amount of unused space in an HDF5 file.

Which compression should I use?

There is no "one size fits all" compression filter solution. One has to consider several characteristics such as the type of data, the desired compression ratio vs. encoding/decoding speed, the general availability of a compression filter, licensing, etc., before committing to a compression filter. This is especially true for HDF5 data producers. The way data is written will affect how much bandwidth consumers will need to download data products, how much system memory and time will be required to read the data and how many data products can be stored on the users system, to name a few.

A comparison of different compression filters is not a subject of this document. However, it is easy to show that, unless a suitable compression method or an advantageous filter combination is chosen, applying the same compression filter to different types of data may not achieve the goal and, for example, reduce HDF5 file size as much as possible.

We looked at one of the NPP EDR product file packaged with it its geolocation information `GCRIO-REDRO_npp_d20030125_t0702533_e0711257_b00993_c20140501163427060570_XXXX_XXX.h5` and used `h5repack` to apply three different compressions to the original file:

1. `Deflate` compression level 7
2. `SZIP` compression using `NN` mode and 32-bit block size
3. Data shuffling in combination with the `deflate` compression level 7

Then we compared the sizes of the 32-bit floating dataset `/All_Data/CrIMSS-EDR-GEO-TC_All/Height` when different types of compression were used, and similarly for the sizes of the 32-bit integer dataset `/All_Data/CrIMSS-EDR_All/FORnum`. The results are shown in Table 1 below.

Data	Original	Deflate level 7	SZIP using NN mode and block-size 32	Shuffle and deflate level 7
32-bit floats	1	2.087	1.628	2.56
32-bit integer	1	3.642	10.832	38.20

Table 1: Compression ratio (CR) for different types of compressions when using h5repack

The combination of the shuffle filter and `deflate` compression level 7 worked well on both floating point and integer datasets, as shown in the fifth column of Table 1. `Deflate` compression worked better than `SZIP` on the floating point dataset, but not on the integer dataset as shown by the results in columns three and four. Clearly, if the objective is to minimize the size of the file, different dataset have to be compressed with different compression methods.

What else can be done to reduce the size of an HDF5 file

Sometimes HDF5 files contain unused space, which can be reduced or eliminated by running `h5repack` without changing any storage parameters of the data. For example, running h5stat on the file `GCRIO-REDRO_npp_d20030125_t0702533_e0711257_b00993_c20140501163524579819_XXXX_XXX.h5` shows:

```
Summary of file space information:
  File metadata: 425632 bytes
  Raw data: 328202 bytes
  Unaccounted space: 449322 bytes
Total space: 1203156 bytes
```

After running `h5repack` one gets a 10-fold reduction in unaccounted space:

```
Summary of file space information:
  File metadata: 425176 bytes
  Raw data: 328202 bytes
  Unaccounted space: 45846 bytes
Total space: 799224 bytes
```

There is also a small reduction in file metadata space.

Conclusion

The most common causes of poor compression in HDF5 files are missing compression filters and use of inadequate (for the data/objectives) compression filters. Inspection of the installed files, HDF5 APIs and HDF5 command-line tools such as `h5dump`, `h5ls`, `h5debug`, `h5stat` and `h5repack` can be used to diagnose and fix most problems.

References

1. The HDF Group. “HDF5 Tutorial”, http://www.hdfgroup.org/HDF5/Tutor/introductory.html
2. The HDF Group. “HDF5 User’s Guide”, Section 5.4.2 http://www.hdfgroup.org/HDF5/doc/UG/index.html
3. The HDF Group. “Filters in HDF5”, http://www.hdfgroup.org/HDF5/doc/H5.user/Filters.html
4. The HDF Group. “HDF5 Dynamically Loaded Filters”, http://www.hdfgroup.org/HDF5/doc/Advanced/DynamicallyLoadedFilters/HDF5DynamicallyLoadedFilters.pdf
5. The HDF Group. “HDF5 Reference Manual”, Property Lists http://www.hdfgroup.org/HDF5/doc/RM/RM_H5P.html#Property-FilterBehavior.
6. The HDF Group, “HDF5 File Format”, Description of the Filter Mask, http://www.hdfgroup.org/HDF5/doc/H5.format.html#V1Btrees
7. The HDF Group. “SZIP compression in HDF products”, http://www.hdfgroup.org/doc_resource/SZIP/

Revision History

May 27, 2014: Version 1 circulated for comment within The HDF Group.

Appendix: Example program

The example program used to create the file discussed in this document, is a modified version of the program available at

http://www.hdfgroup.org/ftp/HDF5/examples/examples-by-api/hdf5-examples/1_8/C/H5D/h5ex_d_gzip.c

It was modified to have chunk dimensions not be factors of the dataset dimensions. Chunk dimensions were chosen for demonstration purposes only and are not recommended for real applications.

```
#include <stdlib.h>

#define FILE            "h5ex_d_gzip.h5"
#define DATASET         "DS1"
#define DIM0            32
#define DIM1            64
#define CHUNK0          5
#define CHUNK1          9

int
main (void)
{
    hid_t           file, space, dset, dcpl;    /* Handles */
    herr_t          status;
    htri_t          avail;
    H5Z_filter_t    filter_type;
    hsize_t         dims[2] = {DIM0, DIM1},
                    chunk[2] = {CHUNK0, CHUNK1};
    size_t          nelmts;
    unsigned int    flags,
                    filter_info;
    int             wdata[DIM0][DIM1],          /* Write buffer */
                    rdata[DIM0][DIM1],          /* Read buffer */
                    max,
                    i, j;


    /*
     * Initialize data.
     */
    for (i=0; i<DIM0; i++)
        for (j=0; j<DIM1; j++)
            wdata[i][j] = i * j - j;

    /*
     * Create a new file using the default properties.
     */
    file = H5Fcreate (FILE, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /*
     * Create dataspace.  Setting maximum size to NULL sets the maximum
     * size to be the current size.
     */
    space = H5Screate_simple (2, dims, NULL);
```

```
    /*
     * Create the dataset creation property list, add the gzip
     * compression filter and set the chunk size.
     */
    dcpl = H5Pcreate (H5P_DATASET_CREATE);
    status = H5Pset_deflate (dcpl, 9);
    status = H5Pset_chunk (dcpl, 2, chunk);

    /*
     * Create the dataset.
     */
    dset = H5Dcreate (file, DATASET, H5T_STD_I32LE, space, H5P_DEFAULT, dcpl,
                H5P_DEFAULT);

    /*
     * Write the data to the dataset.
     */
    status = H5Dwrite (dset, H5T_NATIVE_INT, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                wdata[0]);

    /*
     * Close and release resources.
     */
    status = H5Pclose (dcpl);
    status = H5Dclose (dset);
    status = H5Sclose (space);
    status = H5Fclose (file);

    return 0;
}
```