H5Z-ZFP Documentation

Release 1.1.1

H5Z-ZFP

Aug 29, 2023

Contents

H5Z-ZFP is a compression filter for HDF5 using the ZFP compression library, supporting *lossy* and *lossless* compression of floating point and integer data to meet bitrate, accuracy, and/or precision targets. The filter uses the registered HDF5 filter ID, `32013`. It supports single and double precision floating point and integer data *chunked* in 1, 2 or 3 dimensions. The filter will function on datasets of more than 3 dimensions (or 4 dimensions for ZFP versions 0.5.4 and newer), albeit at the possible expense of compression performance, as long as no more than 3 (or 4) dimensions of the HDF5 dataset chunking are of size greater than 1.

Contents:

CHAPTER 1

Installation

Three ways to install H5Z-ZFP are provided. These are

- With generic (GNU) *make*
- With *CMake*
- With *Spack*

For both generic make and CMake, you are responsible for also installing (or knowing where the installations are) the dependencies, ZFP and HDF5. For Spack installations, Spack will handle installation of dependencies as well.

1.1 Installiang via Generic (GNU) Make

H5Z-ZFP installation supports both vanilla (GNU) Make (described below) as well as *CMake*.

1.1.1 Prerequisites

- ZFP Library (or from Github)
- HDF5 Library
- H5Z-ZFP filter plugin

1.1.2 Compiling ZFP

- There is a `Config` file in top-level directory of the ZFP distribution that holds `make` variables the ZFP Makefiles use. By default, this file is setup for a vanilla GNU compiler. If this is not the appropriate compiler, edit `Config` as necessary to adjust the compiler and compilation flags.
- An important flag you **will** need to adjust in order to use the ZFP library with this HDF5 filter is the `BIT_STREAM_WORD_TYPE` CPP flag. To use ZFP with H5Z-ZFP, the ZFP library **must** be compiled with `BIT_STREAM_WORD_TYPE` of `uint8`. Typically, this is achieved by including a line in `Config` of the form

DEFS += -DBIT_STREAM_WORD_TYPE=uint8. If you attempt to use this filter with a ZFP library compiled differently from this, the filter's can_apply method will always return false. This will result in silently ignoring an HDF5 client's request to compress data with ZFP. Also, be sure to see *Endian Issues*.

- After you have setup Config, simply run make and it will build the ZFP library placing the library in a lib sub-directory and the necessary include files in inc[lude] sub-directory.
- For more information and details, please see the ZFP README.

1.1.3 Compiling HDF5

- If you want to be able to run the fortran tests for this filter, HDF5 must be configured with *both* the --enable-fortran and --enable-fortran2003 configuration switches. Otherwise, any vanilla installation of HDF5 is acceptable.
- The Fortran interface to this filter *requires* a Fortran 2003 compiler because it uses ISO_C_BINDING to define the Fortran interface.
- If you are using HDF5-1.12 and wish to use the filter as a *library* (see *Plugin vs. Library Operation*), you may need configure HDF5 with --disable-memory-alloc-sanity-check to work around a memory management issue in HDF5.

1.1.4 Compiling H5Z-ZFP

H5Z-ZFP is designed to be compiled both as a standalone HDF5 *plugin* and as a separate *library* an application can explicitly link. See *Plugin vs. Library Operation*.

Once you have installed the prerequisites, you can compile H5Z-ZFP using a command-line…

```
make [FC=<Fortran-compiler>] CC=<C-compiler> \
    ZFP_HOME=<path-to-zfp> HDF5_HOME=<path-to-hdf5> \
    PREFIX=<path-to-install>
```

where <path-to-zfp> is a directory containing ZFP inc[lude] and lib dirs and <path-to-hdf5> is a directory containing HDF5 include and lib dirs. If you don't specify a C compiler, it will try to guess one from your path. Fortran compilation is optional. If you do not specify a Fortran compiler, it will not attempt to build the Fortran interface. However, if the variable FC is already defined in your enviornment (as in Spack for example), then H5Z-ZFP will attempt to build Fortran. If this is not desired, the solution is to pass an *empty* FC on the make command line as in…

```
make FC= CC=<C-compiler> \
    ZFP_HOME=<path-to-zfp> HDF5_HOME=<path-to-hdf5> \
    PREFIX=<path-to-install>
```

The Makefile uses GNU Make syntax and is designed to work on OSX and Linux. The filter has been tested on gcc, clang, xlc, icc and pgcc compilers and checked with valgrind.

The command make help will print useful information about various make targets and variables. make check will compile everything and run a handful of tests.

If you don't specify a PREFIX, it will install to ./install. The installed package will look like…

```
$(PREFIX)/include/{H5Zzfp.h,H5Zzfp_plugin.h,H5Zzfp_props.h,H5Zzfp_lib.h}
$(PREFIX)/plugin/libh5zzfp.{so,dylib}
$(PREFIX)/lib/libh5zzfp.a
```

where $(PREFIX) resolves to whatever the full path of the installation is.

To use the installed filter as an HDF5 *plugin*, you would specify, for example, setenv HDF5_PLUGIN_PATH $(PREFIX)/plugin

1.2 Installing via CMake

It is possible to build the H5Z-ZFP filter using the CMake build system. To use CMake for H5Z-ZFP, it is necessary to have also built ZFP with CMake. This is necessary to get the correct dependencies from ZFP. For example, it is possible to build ZFP with OpenMP support. The resulting CMake config files of ZFP build will make sure that this OpenMP dependency is correctly propagated to the build of H5Z-ZFP filter. However, for HDF5 it is not necessary to build it with its CMake build system but it is strongly recommended.

ZFP must have been *configured* with BIT_STREAM_WORD_TYPE of uint8 as described above.

Similar as for the Makefile installation, the CMake build system is designed such it compiles both the standalone HDF5 *plugin* and a separate *library* an application can explicitly link. See *Plugin vs. Library Operation*

Once both HDF5 and ZFP have been installed, H5Z-ZFP can be compiled using a command=line…

```
export HDF5_DIR=<path-to_hdf5>
export ZFP_DIR=<path-to-zfp-config>
CC=<C-compiler> FC=<Fortran-compiler> cmake -DCMAKE_INSTALL_PREFIX=<path-to-install>
↪<src-dir>
```

where <path-to-zfp-config> is a directory containing zfp-config.cmake and <path-to-hdf5> is a directory containing HDF5 include and lib directories. Furthermore, src-dir is the directory where the H5Z-ZFP source is located and path-to-install is the directory in which the resulting *plugin* and *library* will be installed. Once cmake has finished successfully, you can build and install the filter using the command…

```
make install
```

This cmake and make combination builds both the C and Fortran interface. In the case you want to specify the <path-to-hdf5> and <path-to-zfp>> via command-line to CMake, the command looks like this…

```
CC=<C-compiler> FC=<Fortran-compiler> cmake -DCMAKE_INSTALL_PREFIX=<path-to-install>
    -DCMAKE_PREFIX_PATH="<path-tohdf5>;<path-to-zfp>" <src-dir>
```

Note: The double quotes in the CMAKE_PREFIX_PATH expression are necessary to make sure that semicolon is interpreted as a semicolon instead of a new command.

It is possible to build the filter without the Fortran interface. This is done as follows…

```
export HDF5_DIR=<path-to_hdf5>
export ZFP_DIR=<path-to-zfp>
CC=<C-compiler> cmake -DCMAKE_INSTALL_PREFIX=<path-to-install> -DFORTRAN_
↪INTERFACE:BOOL=OFF <src-dir>
```

followed by the same make command…

```
make install
```

1.3 Including H5Z-ZFP filter in a CMake project

Suppose you have built the H5Z-ZFP filter using the CMake build system and installed it in `<path-to-h5z_zfp>`. To include it in another CMake project is done using the following steps. First edit the `CMakeLists.txt` by adding the following two lines...

```
cmake_policy(SET CMP0028 NEW) # Double colon in target name means ALIAS or IMPORTED target.
...
set(H5Z_ZFP_USE_STATIC_LIBS OFF)
find_package(H5Z_ZFP 1.0.1 CONFIG)
...
target_link_libraries(<target> h5z_zfp::h5z_zfp)
...
```

where `<target>` in the target within the CMake project. This could be, for example, an executable or library. Furthermore, check if the `cmake` version is equal or greater than 3.9. Next, you need to make sure that the filter can be found by CMake, followed by `cmake` itself and `make`...

```
export H5Z_ZFP_DIR=<path-to-h5z_zfp>
CC=<C-compiler> cmake -DCMAKE_INSTALL_PREFIX=<path-to-install> <src-dir>
make install
```

The `cmake` command itself could be different depending on the CMake project you have created. If you want to make use of the H5Z-ZFP *library* instead of the plugin, change cmake variable `H5Z_ZFP_USE_STATIC_LIBS` to `ON` and build the project.

1.4 Installing via Spack

If you already have experience with Spack, one way to install H5Z-ZFP is to use the command `spack install h5z-zfp`. If you do not have Spack installed, it is easy to install. Assuming you are working in a Bash shell...:

```
git clone https://github.com/llnl/spack.git
. spack/share/spack/setup-env.sh
spack install h5z-zfp
```

If you are using a version of Spack very much older than the release of H5Z-ZFP you intend to use, you may have to *pin* various versions of H5Z-ZFP, ZFP and/or HDF5. This is done by using Spack's `@` modifier to specify versions. For example, to *pin* the version of the ZFP library to 0.5.5, the Spack command would look like:

```
spack install h5z-zfp ^zfp@0.5.5
```

To use the `develop` version of H5Z-ZFP with version 1.10.6 of HDF5

```
spack install h5z-zfp@develop ^hdf5@1.10.6
```

By default, H5Z-ZFP will attempt to build with Fortran support which requires a Fortran compiler. If you wish to exclude support for Fortran, use the command:

```
spack install h5z-zfp ~fortran
```

Note: These commands will build H5Z-ZFP **and** all of its dependencies including the HDF5 library *as well as a number of other dependencies you may not initially expect*. Be patient and let the build complete. It may take more than an hour.

In addition, by default, Spack installs packages to directory *hashes within* the cloned Spack repository's directory tree, `$spack/opt/spack`. You can find the resulting installed HDF5 library with the command `spack find -vp hdf5` and the resulting H5Z-ZFP plugin installation with the command `spack find -vp h5z-zfp`. If you wish to exercise more control over where Spack installs things, have a look at configuring Spack

1.5 H5Z-ZFP Source Code Organization

The source code is in two separate directories

- `src` includes the ZFP filter and a few header files
  - `H5Zzfp_plugin.h` is an optional header file applications *may* wish to include because it contains several convenient macros for easily controlling various compression modes of the ZFP library (*rate*, *precision*, *accuracy*, *expert*) via the *Generic Interface*.
  - `H5Zzfp_props.h` is a header file that contains functions to control the filter using *temporary Properties Interface*. Fortran callers are *required* to use this interface.
  - `H5Zzfp_lib.h` is a header file for applications that wish to use the filter explicitly as a library rather than a plugin.
  - `H5Zzfp.h` is an *all-of-the-above* header file for applications that don't care too much about separating out the above functionalities.
- `test` includes various tests. In particular `test_write.c` includes examples of using both the *Generic Interface* and *Properties Interface*. In addition, there is an example of how to use the filter from Fortran in `test_rw_fortran.F90`.

1.6 Silo Integration

This filter (`H5Zzfp.c`) is also built-in to the Silo library. In particular, the ZFP library itself is also embedded in Silo but is protected from appearing in Silo's global namespace through a struct of function pointers (see Namespaces in C). If you happen to examine the source code here for H5Z-ZFP, you will see some logic here that is specific to using this plugin within Silo and dealing with ZFP as an embedded library using this struct of function pointers wrapper. In the source code for H5Z-ZFP this manifests as something like what is shown in the code snippet below...

```
    switch (ndims_used)
    {
        case 1: dummy_field = Z zfp_field_1d(0, zt, dims_used[0]); break;
        case 2: dummy_field = Z zfp_field_2d(0, zt, dims_used[1], dims_used[0]); break;
        case 3: dummy_field = Z zfp_field_3d(0, zt, dims_used[2], dims_used[1], dims_used[0]); break;
#if ZFP_VERSION_NO >= 0x0540
        case 4: dummy_field = Z zfp_field_4d(0, zt, dims_used[3], dims_used[2], dims_used[1], dims_used[0]); break;
#endif
#if ZFP_VERSION_NO < 0x0530
        default: H5Z_ZFP_PUSH_AND_GOTO(H5E_PLINE, H5E_BADVALUE, 0,
                     "chunks may have only 1...3 non-unity dims");
#else
        default: H5Z_ZFP_PUSH_AND_GOTO(H5E_PLINE, H5E_BADVALUE, 0,
```

(continues on next page)

(continued from previous page)

```
                        "chunks may have only 1...4 non-unity dims");
#endif
    }
```

In the code snippet above, note the funny `Z` in front of calls to various methods in the ZFP library. When compiling H5Z-ZFP normally, that `Z` normally resolves to the empty string. But, when the code is compiled with `-DAS_SILO_BUILTIN` (which is supported and should be done *only* when `H5Zzfp.c` is being compiled *within* the Silo library and *next to* a version of ZFP that is embedded in Silo) that `Z` resolves to the name of a struct and struct-member dereferncing operator as in `zfp.`. There is a similar `B` used for a similar purpose ahead of calls to ZFP's bitstream library. This is something to be aware of and to adhere to if you plan to contribute any code changes here.

CHAPTER 2

Interfaces

There are two interfaces to control the filter. One uses HDF5's *generic* interface via an array of `unsigned int cd_values` as is used in H5Pset_filter(). The other uses HDF5 property lists added to the dataset creation property list used when the dataset to be compressed is being created. You can find examples of writing HDF5 data using both the generic and properties interfaces in test_write.c.

The filter itself supports either interface. The filter also supports all of the standard ZFP controls for affecting compression including *rate*, *precision*, *accuracy*, *expert* and *reversible* modes. For more information and details about these modes of controlling ZFP compression, please see the ZFP README.

Finally, you should *not* attempt to combine the ZFP filter with any other *byte order altering* filter such as, for example, HDF5's shuffle filter. Space-performance will be ruined. This is in contrast to HDF5's deflate filter which often performs *better* when used in conjunction with the shuffle filter. To understand why, see the description of *endian issues*.

2.1 Generic Interface

The generic interface is the only means of controlling the H5Z-ZFP filter when it is used as a dynamically loaded HDF5 plugin.

For the generic interface, the following CPP macros are defined in the `H5Zzfp_plugin.h` header file:

```
H5Pset_zfp_rate_cdata(double rate, size_t cd_nelmts, unsigned int *cd_vals);
H5Pset_zfp_precision_cdata(unsigned int prec, size_t cd_nelmts, unsigned int *cd_
↪vals);
H5Pset_zfp_accuracy_cdata(double acc, size_t cd_nelmts, unsigned int *cd_vals);
H5Pset_zfp_expert_cdata(unsigned int minbits, unsigned int maxbits,
                        unsigned int maxprec, int minexp,
                        size_t cd_nelmts, unsigned int *cd_vals);
H5Pset_zfp_reversible_cdata(size_t cd_nelmts, unsigned int *cd_vals);
```

These macros utilize *type punning* to store the relevant ZFP parameters into a sufficiently large array (>=6) of `unsigned int cd_values`. It is up to the caller to then call H5Pset_filter() with the array of cd_values constructed by one of these macros.

Here is example code from test_write.c...

```
    if (zfpmode == H5Z_ZFP_MODE_RATE)
        H5Pset_zfp_rate_cdata(rate, cd_nelmts, cd_values);
    else if (zfpmode == H5Z_ZFP_MODE_PRECISION)
        H5Pset_zfp_precision_cdata(prec, cd_nelmts, cd_values);
    else if (zfpmode == H5Z_ZFP_MODE_ACCURACY)
        H5Pset_zfp_accuracy_cdata(acc, cd_nelmts, cd_values);
    else if (zfpmode == H5Z_ZFP_MODE_EXPERT)
        H5Pset_zfp_expert_cdata(minbits, maxbits, maxprec, minexp, cd_nelmts, cd_
↪values);
    else if (zfpmode == H5Z_ZFP_MODE_REVERSIBLE)
        H5Pset_zfp_reversible_cdata(cd_nelmts, cd_values);
    else
        cd_nelmts = 0; /* causes default behavior of ZFP library */

    /* print cd-values array used for filter */
    printf("\n%d cd_values=", (int) cd_nelmts);
    for (int i = 0; i < (int) cd_nelmts; i++)
        printf("%u,", cd_values[i]);
    printf("\n");

    /* Add filter to the pipeline via generic interface */
    if (0 > H5Pset_filter(cpid, H5Z_FILTER_ZFP, H5Z_FLAG_MANDATORY, cd_nelmts, cd_
↪values)) SET_ERROR(H5Pset_filter);
```

However, these macros are only a convenience. You do not **need** the `H5Zzfp_plugin.h` header file if you want to avoid using it. But, you are then responsible for setting up the `cd_values` array correctly for the filter. For reference, the `cd_values` array for this ZFP filter is defined like so...

	cd_values index					
ZFP mode	0	1	2	3	4	5
rate	1	unused	rateA	rateB	unused	unused
precision	2	unused	prec	unused	unused	unused
accuracy	3	unused	accA	accB	unused	unused
expert	4	unused	minbits	maxbits	maxprec	minexp
reversible	5	unused	unused	unused	unused	unsued

A/B are high/low 32-bit words of a double.

Note that the cd_values used in the generic interface to `H5Pset_filter()` are **not the same** cd_values ultimately stored to the HDF5 dataset header for a compressed dataset. The values are transformed in the set_local method to use ZFP's internal routines for 'meta' and 'mode' data. So, don't make the mistake of examining the values you find in a file and think you can use those same values, for example, in an invokation of h5repack.

2.2 Properties Interface

For the properties interface, the following functions are defined in the `H5Zzfp_props.h` header file:

```
herr_t H5Pset_zfp_rate(hid_t dcpl_id, double rate);
herr_t H5Pset_zfp_precision(hid_t dcpl_id, unsigned int prec);
herr_t H5Pset_zfp_accuracy(hid_t dcpl_id, double acc);
herr_t H5Pset_zfp_expert(hid_t dcpl_id,
```

(continues on next page)

(continued from previous page)

```
    unsigned int minbits, unsigned int maxbits,
    unsigned int maxprec, int minexp);
herr_t H5Pset_zfp_reversible(hid_t dcpl_id);
```

These functions take a dataset creation property list, `hid_t dcp_lid` and create temporary HDF5 property list entries to control the ZFP filter. Calling any of these functions removes the effects of any previous call to any one of these functions. In addition, calling any one of these functions also has the effect of adding the filter to the pipeline.

Here is example code from test_write.c...

```
    H5Z_zfp_initialize();

    /* Setup the filter using properties interface. These calls also add
       the filter to the pipeline */
    if (zfpmode == H5Z_ZFP_MODE_RATE)
        H5Pset_zfp_rate(cpid, rate);
    else if (zfpmode == H5Z_ZFP_MODE_PRECISION)
        H5Pset_zfp_precision(cpid, prec);
    else if (zfpmode == H5Z_ZFP_MODE_ACCURACY)
        H5Pset_zfp_accuracy(cpid, acc);
    else if (zfpmode == H5Z_ZFP_MODE_EXPERT)
        H5Pset_zfp_expert(cpid, minbits, maxbits, maxprec, minexp);
    else if (zfpmode == H5Z_ZFP_MODE_REVERSIBLE)
        H5Pset_zfp_reversible(cpid);

```

The properties interface is more type-safe than the generic interface. However, there is no way for the implementation of the properties interface to reside within the filter plugin itself. The properties interface requires that the caller link with with the filter as a *library*, `libh5zzfp.a`. The generic interface does not require this.

Note that either interface can be used whether the filter is used as a plugin or as a library. The difference is whether the application calls `H5Z_zfp_initialize()` or not.

2.3 Fortran Interface

Fortran equivalents for both the properties and generic interfaces, described above, has been added by Scot Breitenfeld of the HDF5 group. The code that implements the Fortran interfaces is in the file `H5Zzfp_props_f.F90`. An example of its use is in `test/test_rw_fortran.F90`.

2.4 Plugin vs. Library Operation

The filter is designed to be compiled for use as both a standalone HDF5 dynamically loaded HDF5 plugin and as an explicitly linked *library*. When it is used as a plugin, it is a best practice to link the ZFP library into the plugin dynamic/shared object as a *static* library. Why? In so doing, we ensure that all ZFP public namespace symbols remain *confined* to the plugin so as not to interfere with any application that may be directly explicitly linking to the ZFP library for other reasons.

All HDF5 applications are *required* to *find* the plugin dynamic library (named `lib*.{so,dylib}`) in a directory specified by the enviornment variable, `HDF5_PLUGIN_PATH`. Currently, the HDF5 library offers no mechanism for applications themselves to have pre-programmed paths in which to search for a plugin. Applications are then always vulnerable to an incorrectly specified or unspecified `HDF5_PLUGIN_PATH` environment variable.

However, the plugin can also be used explicitly as a *library*. In this case, **do not** specify the `HDF5_PLUGIN_PATH` enviornment variable and instead have the application link to `libH5Zzfp.a` in the `lib` dir of the installation. Instead two initialization and finalization routines are defined:

```
int H5Z_zfp_initialize(void);
int H5Z_zfp_finalize(void);
```

These functions are defined in the `H5Zzfp_lib.h` header file. Any applications that wish to use the filter as a *library* are required to call the initialization routine, `H5Z_zfp_initialize()` before the filter can be referenced. In addition, to free up resources used by the filter, applications may call `H5Z_zfp_finalize()` when they are done using the filter.

CHAPTER 3

HDF5 Chunking

HDF5's dataset chunking feature is a way to optimize data layout on disk to support partial dataset reads by downstream consumers. This is all the more important when compression filters are applied to datasets as it frees a consumer from suffering the UNcompression of an entire dataset only to read a portion.

3.1 ZFP Chunklets

When using HDF5 chunking with ZFP compression, it is important to account for the fact that ZFP does its work in tiny 4^d chunklets of its own where *d* is the dataset dimension (*rank* in HDF5 parlance). This means that whenever possible, the chunking dimensions you select in HDF5 should be multiples of 4. When a chunk dimension is not a multiple of 4, ZFP will wind up with partial chunklets, which will be padded with useless data, reducing the results' overall time and space efficiency.

The degree to which this may degrade performance depends on the percentage of a chunk that is padded. Suppose we have a 2D chunk of dimensions 27 x 101. ZFP will have to treat it as 28 x 104 by padding out each dimension to the next closest multiple of 4. The fraction of space that will wind up being wasted due to ZFP chunklet padding will be (28x104-27x101) / (28x104), which is about 6.4%. On the other hand, consider a 3D chunk that is 1024 x 1024 x 2. ZFP will have to treat it as a 1024 x 1024 x 4 resulting in 50% waste.

The latter example is potentially very relevant when applying ZFP to compress data along the *time* dimension in a large, 3D, simulation. Ordinarily, a simulation advances one time step at a time and so needs to store in memory only the *current* timestep. However, in order to give ZFP enough *width* in the time dimension to satisfy the minimum chunklet dimension size of 4, the simulation needs to keep in memory 4 timesteps. This is demonstrated in the example below.

3.2 Partial I/O Requests

In any given H5Dwrite call, the caller has the option of writing (or reading) only a portion of the data in the dataset. This is a *partial I/O* request. This is handled by the `mem_space_id` and `file_space_id` arguments in an H5Dwrite call.

An HDF5 producer or consumer can issue partial I/O requests on *any* HDF5 dataset regardless of whether the dataset is compressed or not or whether the dataset has `H5D_CONTIGUOUS` layout. When combining partial I/O with compression, chunk size and shape in relation to partial I/O request size and shape will have an impact on performance.

This is particularly important in *writer* scenarios if an I/O request winds up overlapping chunks only partially. Suppose the partially overlapped chunks exist in the file (from a previous write, for example). In that case, the HDF5 library may wind up having to engage in *read-modify-write* operations for those chunks.

If the partially overlapped chunks do not exist in the file, the HDF5 library will wind up *fill-value* padding the chunks before they are written. HDF5's default fill value is zero (as defined by the associated datatype). Data producers can choose the desired fill value (see H5Pset_fill_value) for a dataset, but this fill value can impact the space-performance of the compression filter. On the other hand, if the partial chunks in one I/O request wind up getting fully filled in another, any fill value impacts on compressor performance are resolved.

Finally, HDF5 manages a chunk cache and data sieving buffer to help alleviate some of the I/O performance issues that can be encountered in these situations.

3.3 More Than 3 (or 4) Dimensions

Versions of ZFP 0.5.3 and older support compression in only 1,2 or 3 dimensions. Versions of ZFP 0.5.4 and newer also support 4 dimensions.

What if you have a dataset with more dimensions than ZFP can compress? You can still use the H5Z-ZFP filter. But, in order to do so, you are *required* to chunk the dataset[1] . Furthermore, you must select a chunk size such that no more than 3 (or 4 for ZFP 0.5.4 and newer) dimensions are non-unitary (e.g. of size one).

For example, what if you are using ZFP 0.5.3 and have a 4D HDF5 dataset you want to compress? To do this, you will need to chunk the dataset and when you define the chunk size and shape, you will need to select which of the 4 dimensions of the chunk you do *not* intend to have ZFP perform compression along by setting the size of the chunk in that dimension to unity (1). When you do this, as HDF5 processes writes and reads, it will organize the data so that all the H5Z-ZFP filter *sees* are chunks which have *extent* only in the non-unity dimensions of the chunk.

In the example below, we have a 4D array of shape `int dims[] = {256,128,32,16};` that we have intentionally constructed to be *smooth* in only 2 of its 4 dimensions (e.g. correlation is high in those dimensions). Because of that, we expect ZFP compression to do well along those dimensions, and we do not want ZFP to compress along the other 2 dimensions. The *uncorrelated* dimensions here are dimensions with indices `1` (`128` in `dims[]`) and `3` (`16` in `dims[]`). Thus, our chunk size and shape are chosen to set the size for those dimension indices to `1`, `hsize_t hchunk[] = {256,1,32,1};`

```
    if (highd)
    {
     /* dimension indices 0   1   2  3 */
        int dims[] = {256,128,32,16};
        int ucdims[]={1,3}; /* UNcorrleted dimensions indices */
        hsize_t hdims[] = {256,128,32,16};
        hsize_t hchunk[] = {256,1,32,1};

        buf = gen_random_correlated_array(TYPDBL, 4, dims, 2, ucdims);

        cpid = setup_filter(4, hchunk, zfpmode, rate, acc, prec, minbits, maxbits, maxprec, minexp);

        if (0 > (sid = H5Screate_simple(4, hdims, 0))) SET_ERROR(H5Screate_simple);

```

(continues on next page)

[1] The HDF5 library currently requires dataset chunking anyways for any dataset that has any kind of filter applied.

(continued from previous page)

```
        /* write the data WITHOUT compression */
        if (0 > (dsid = H5Dcreate(fid, "highD_original", H5T_NATIVE_DOUBLE, sid, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT))) SET_ERROR(H5Dcreate);
        if (0 > H5Dwrite(dsid, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf)) SET_ERROR(H5Dwrite);
        if (0 > H5Dclose(dsid)) SET_ERROR(H5Dclose);

        /* write the data with compression */
        if (0 > (dsid = H5Dcreate(fid, "highD_compressed", H5T_NATIVE_DOUBLE, sid, H5P_DEFAULT, cpid, H5P_DEFAULT))) SET_ERROR(H5Dcreate);
        if (0 > H5Dwrite(dsid, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf)) SET_ERROR(H5Dwrite);
        if (0 > H5Dclose(dsid)) SET_ERROR(H5Dclose);

        /* clean up from high dimensional test */
        if (0 > H5Sclose(sid)) SET_ERROR(H5Sclose);
        if (0 > H5Pclose(cpid)) SET_ERROR(H5Pclose);
        free(buf);
    }
```

What analysis process should you use to select the chunk shape? Depending on what you expect in the way of access patterns in downstream consumers, this can be a challenging question to answer. There are potentially two competing interests. One is optimizing the chunk size and shape for access patterns anticipated by downstream consumers. The other is optimizing the chunk size and shape for compression. These two interests may not be compatible and you may have to compromise between them. We illustrate the issues and tradeoffs using an example.

3.4 Compression *Along* the *State Iteration* Dimension

By *state iteration* dimension, we refer to the data producer's main iteration loop(s). For example, the main iteration dimension for many PDE-based simulations is *time*. But, for some *outer loop* methods, the main iteration dimension(s) might be some kind of parameter study including multiple parameters.

The challenge here is to manage the data to meet ZFP's chunklet size and shape *minimum* requirements. In any H5Dwrite at least 4 *samples* along a ZFP compression dimension are needed, or there will be wasted space due to padding. This means that data must be *buffered* along those dimensions *before* H5Dwrite's can be issued.

For example, suppose you have a tensor-valued field (e.g. a 3x3 matrix at every *point*) over a 4D (3 spatial dimensions and 1 time dimension), regularly sampled domain? Conceptually, this is a 6 dimensional dataset in HDF5 with one of the dimensions (the *time* dimension) *extendible*. So, you are free to define this as a 6 dimensional dataset in HDF5. But, you will also have to chunk the dataset. You can select any chunk shape you want, except that no more than 3 (or 4 for ZFP versions 0.5.4 and newer) dimensions of the chunk can be non-unity.

In the code snippet below, we demonstrate this case. A key issue to deal with is that because we will use ZFP to compress along the time dimension, this forces us to keep in memory a sufficient number of timesteps to match ZFP's chunklet size of 4.

The code below iterates over 9 timesteps. Each of the first two groups of 4 timesteps are buffered in memory in `tbuf`. Once 4 timesteps have been buffered, we can issue an H5Dwrite call doing hyperslab can issue an H5Dwrite call doing hyperslab partial I/O on the 6D, extendible dataset. But, notice that the chunk dimensions (line 10) are such that only 4 of the 6 dimensions are non-unity. This means ZFP will only ever see something to compress that is essentially 4D.

On the last iteration, we have only one *new* timestep. So, when we write this to ZFP 75% of that write will be *wasted* due to ZFP chunklet padding. However, if the application were to *restart* from this time and continue forward, this *waste* would ultimately get overwritten with new timesteps.

```
    /* Test six dimensional, time varying array...
           ...a 3x3 tensor valued variable
           ...over a 3D+time domain.
           Dimension sizes are chosen to miss perfect ZFP block alignment.
    */
    if (sixd)
    {
        void *tbuf;
        int t, dims[] = {31,31,31,3,3}; /* a single time instance */
        int ucdims[]={3,4}; /* indices of UNcorrleted dimensions in dims (tensor components) */
        hsize_t hdims[] = {31,31,31,3,3,H5S_UNLIMITED};
        hsize_t hchunk[] = {31,31,31,1,1,4}; /* 4 non-unity, requires >= ZFP 0.5.4 */
        hsize_t hwrite[] = {31,31,31,3,3,4}; /* size/shape of any given H5Dwrite */

        /* Setup the filter properties and create the dataset */
        cpid = setup_filter(6, hchunk, zfpmode, rate, acc, prec, minbits, maxbits, maxprec, minexp);

        /* Create the time-varying, 6D dataset */
        if (0 > (sid = H5Screate_simple(6, hwrite, hdims))) SET_ERROR(H5Screate_simple);
        if (0 > (dsid = H5Dcreate(fid, "6D_extendible", H5T_NATIVE_DOUBLE, sid, H5P_DEFAULT, cpid, H5P_DEFAULT))) SET_ERROR(H5Dcreate);
        if (0 > H5Sclose(sid)) SET_ERROR(H5Sclose);
        if (0 > H5Pclose(cpid)) SET_ERROR(H5Pclose);

        /* Generate a single buffer which we'll modulate by a time-varying function
           to represent each timestep */
        buf = gen_random_correlated_array(TYPDBL, 5, dims, 2, ucdims);

        /* Allocate the "time" buffer where we will buffer up each time step
           until we have enough to span a width of 4 */
        tbuf = malloc(31*31*31*3*3*4*sizeof(double));

        /* Iterate, writing 9 timesteps by buffering in time 4x. The last
           write will contain just one timestep causing ZFP to wind up
           padding all those blocks by 3x along the time dimension.  */
        for (t = 1; t < 10; t++)
        {
            hid_t msid, fsid;
            hsize_t hstart[] = {0,0,0,0,0,t-4}; /* size/shape of any given H5Dwrite */
            hsize_t hcount[] = {31,31,31,3,3,4}; /* size/shape of any given H5Dwrite */
            hsize_t hextend[] = {31,31,31,3,3,t}; /* size/shape of */

            /* Update (e.g. modulate) the buf data for the current time step */
            modulate_by_time(buf, TYPDBL, 5, dims, t);

            /* Buffer this timestep in memory. Since chunk size in time dimension is 4,
               we need to buffer up 4 time steps before we can issue any writes */
            buffer_time_step(tbuf, buf, TYPDBL, 5, dims, t);

            /* If the buffer isn't full, just continue updating it */
            if (t%4 && t!=9) continue;

```

(continues on next page)

(continued from previous page)

```
            /* For last step, adjust time dim of this write down from 4 to just 1 */
            if (t == 9)
            {
                /* last timestep, write a partial buffer */
                hwrite[5] = 1;
                hcount[5] = 1;
            }

            /* extend the dataset in time */
            if (t > 4)
                H5Dextend(dsid, hextend);

            /* Create the memory dataspace */
            if (0 > (msid = H5Screate_simple(6, hwrite, 0))) SET_ERROR(H5Screate_simple);

            /* Get the file dataspace to use for this H5Dwrite call */
            if (0 > (fsid = H5Dget_space(dsid))) SET_ERROR(H5Dget_space);

            /* Do a hyperslab selection on the file dataspace for this write*/
            if (0 > H5Sselect_hyperslab(fsid, H5S_SELECT_SET, hstart, 0, hcount, 0)) SET_ERROR(H5Sselect_hyperslab);

            /* Write this iteration to the dataset */
            if (0 > H5Dwrite(dsid, H5T_NATIVE_DOUBLE, msid, fsid, H5P_DEFAULT, tbuf)) SET_ERROR(H5Dwrite);
            if (0 > H5Sclose(msid)) SET_ERROR(H5Sclose);
            if (0 > H5Sclose(fsid)) SET_ERROR(H5Sclose);
        }
        if (0 > H5Dclose(dsid)) SET_ERROR(H5Dclose);
        free(buf);
        free(tbuf);
    }
```

CHAPTER 4

H5Z-ZFP and the HDF5 filter's cd_values

Note: The details described here are likely relevant only to *developers* of the H5Z-ZFP filter. If you just want to *use* the filter, you can ignore this material.

The HDF5 library uses an array of values, named `cd_values` in formal arguments documenting various API functions, for managing *auxiliary data* for a filter. Instances of this `cd_values` array are used in two subtly different ways within HDF5.

The first use is in *passing* auxiliary data for a filter from the caller to the library when initially creating a dataset. This happens *directly* in an `H5Pset_filter()` (see here) call.

The second use is in *persisting* auxiliary data for a filter to the dataset's object *header* in a file. This happens *indirectly* as part of an `H5Dcreate()` call.

When a dataset creation property list includes a filter, the filter's `set_local()` method is called (see H5Zregister()) as part of the `H5Dcreate` call. In the filter's `set_local()` method, the `cd_values` that were *passed* by the caller (in `H5Pset_filter()`) are often modified (via `H5Pmodify_filter()` (see here) before they are *persisted* to the dataset's object header in a file.

Among other things, this design allows a filter to be generally configured for *any* dataset in a file and then adjusted as necessary to handle such things as data type and/or dimensions when it is applied to a specific dataset. Long story short, the data stored in `cd_values` of the dataset object's header in the file are often not the same values passed by the caller when the dataset was created.

To make matters a tad more complex, the `cd_values` data is treated by HDF5 as an array of C typed, 4-byte, `unsigned integer` values. Furthermore, regardless of endianness of the data producer, the persisted values are always stored in little-endian format in the dataset object header in the file. Nonetheless, if the persisted `cd_values` data is ever retrieved (e.g. via `H5Pget_filter_by_id()` (see here), the HDF5 library ensures the data is returned to callers with proper endianness. When command-line tools like `h5ls` and `h5dump` print `cd_values`, the data will be displayed correctly.

Handling double precision auxiliary data via `cd_values` is still more complicated because a single double precision value will span multiple entries in `cd_values` in almost all cases. Setting aside the possibility of differing floating point formats between the producer and consumers, any endianness handling the HDF5 library does for the 4-byte entries in `cd_values` will certainly not ensure proper endianness handling of larger values. It is impossible for command-line tools like `h5ls` and `h5dump` to display such data correctly.

Fortunately, the ZFP library has already been designed to handle these issues as part of the ZFP's *native* stream header. But, the ZFP library handles these issues in an endian-agnostic way. Consequently, the H5Z-ZFP filter uses the `cd_values` that is persisted to a dataset's object header to store ZFP's stream header. ZFP's stream header is stored starting at `&cd_values[1]`. `cd_values[0]` is used to stored H5Z-ZFP filter and ZFP library and ZFP encoder version information.

This also means that H5Z-ZFP avoids the overhead of duplicating the ZFP stream header in each dataset chunk. For larger chunks, these savings are probably not too terribly significant.

CHAPTER 5

Direct Writes (and Reads)

The purpose of direct writes is to enable an application to write data that is already compressed in memory *directly* to an HDF5 file without first uncompressing it so the filter can then turn around and compress it during write. However, once data is written to the file with a *direct* write, consumers must still be able to read it without concern for how the producer wrote it.

Doing this requires the use of an advanced HDF5 function for direct writes.

At present, we demonstrate only minimal functionality here using *single chunking*, where the chunk size is chosen to match the size of the entire dataset. To see an example of code that does this, have a look at…

```
    if (zfparr>0 && zfpmode==1 && rate>0)
    {
        int             dims[] = {38, 128};
       /*int       chunk_dims[] = {19, 34};*/
        int       chunk_dims[] = {38, 128};
        hsize_t        hdims[] = {38, 128};
       /*hsize_t hchunk_dims[] = {19, 34};*/
        hsize_t hchunk_dims[] = {38, 128};
        hsize_t hchunk_off[] = {0, 0};
#if defined(ZFP_LIB_VERSION) && ZFP_LIB_VERSION<=0x055
        cfp_array2d *origarr;
#else
        cfp_array2d origarr;
#endif

        /* Create the array data */
        buf = gen_random_correlated_array(TYPDBL, 2, dims, 0, 0);

        /* Instantiate a cfp array */
        origarr = cfp.array2d.ctor(dims[1], dims[0], rate, buf, 0);
        cfp.array2d.flush_cache(origarr);

        cpid = setup_filter(2, hchunk_dims, 1, rate, acc, prec, minbits, maxbits, maxprec, minexp);

```

(continues on next page)

(continued from previous page)

```
        if (0 > (sid = H5Screate_simple(2, hdims, 0))) SET_ERROR(H5Screate_simple);

        /* write the data WITHOUT compression */
        if (0 > (dsid = H5Dcreate(fid, "zfparr_original", H5T_NATIVE_DOUBLE, sid, H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT))) SET_ERROR(H5Dcreate);
        if (0 > H5Dwrite(dsid, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf)) SET_ERROR(H5Dwrite);
        if (0 > H5Dclose(dsid)) SET_ERROR(H5Dclose);

        /* write the data with compression via the filter */
        if (0 > (dsid = H5Dcreate(fid, "zfparr_compressed", H5T_NATIVE_DOUBLE, sid, H5P_DEFAULT, cpid, H5P_DEFAULT))) SET_ERROR(H5Dcreate);
        if (0 > H5Dwrite(dsid, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL, H5P_DEFAULT, buf)) SET_ERROR(H5Dwrite);
        if (0 > H5Dclose(dsid)) SET_ERROR(H5Dclose);

        /* write the data direct from compressed array using H5Dwrite_chunk calls */
        if (0 > (dsid = H5Dcreate(fid, "zfparr_direct", H5T_NATIVE_DOUBLE, sid, H5P_DEFAULT, cpid, H5P_DEFAULT))) SET_ERROR(H5Dcreate);
        if (0 > H5Dwrite_chunk(dsid, H5P_DEFAULT, 0, hchunk_off, cfp.array2d.compressed_size(origarr), cfp.array2d.compressed_data(origarr))) SET_ERROR(H5Dwrite_chunk);

        if (0 > H5Dclose(dsid)) SET_ERROR(H5Dclose);

        free(buf);
        cfp.array2d.dtor(origarr);
    }
```

In particular, look for the line using `H5Dchunk_write` in place of `H5Dwrite`. In all other respects, the code looks the same.

The test case for this code writes uncompressed data as a dataset named `zfparr_original`, the compressed dataset named `zfparr_compressed` using the filter and then the compressed data a second time named `zfparr_direct` using a direct write. Then, the `h5diff` tool is used to compare the data in the original and the direct write datasets.

Note that in order for consumers to work as normal, the producer must set dataset *creation* properties as it ordinarily would using the H5Z-ZFP filter. In the call to `H5Dchunk_write`, the caller indicates to the HDF5 library not to invoke the filter via the `filters` mask argument.

CHAPTER 6

Using H5Z-ZFP Plugin with H5Repack

A convenient way to use and play with the ZFP filter is as a *plugin* with the HDF5 h5repack utility using the `-f` command-line argument to apply ZFP to existing data in a file.

6.1 Patching h5repack

Warning: Versions of HDF5's `h5repack` utility prior to 1.10.4 contain a bug that prevents proper parsing of the `-f` argument's option. In order to use `h5repack` with `-f` argument as described here, you need to apply the patch from h5repack_parse.patch. To do so, after you've downloaded and untar'd HDF5 but before you've built it, do something like the following using HDF5-1.8.14 as an example:

```
gunzip < hdf5-1.8.14.tar.gz | tar xvf -
cd hdf5-1.8.14
patch ./tools/h5repack/h5repack_parse.c <path-to-H5Z-ZFP-test-dir>/h5repack_parse.
↪patch
```

6.2 Constructing an HDF5 cd_values array

HDF5's `h5repack` utility uses only the *generic* interface to HDF5 filters. Another challenge in using `h5repack` as described here is constructing the set `unsigned int cd_values` as is used in H5Pset_filter() required by the *generic* HDF5 filter interface, especially because of the type-punning (doubles as unsigned int) which may be involved.

Note: Querying an existing dataset using `h5dump` or `h5ls` to obtain the `cd_values` *stored* with a ZFP compressed dataset will not provide the correct `cd_values` needed to invoke the filter. This is because the `cd_values` stored in the file are different from those needed in the *generic* interface to invoke the ZFP filter.

To facilitate constructing a valid `-f` argument to `h5repack`, there is a utility program, `print_h5repack_farg`, which is presently in the `test` directory and is built when tests are built. This program was originally written simply to facilitate testing of H5Z-ZFP filter. It should eventually be made a first class *tool* installed with H5Z-ZFP. However, it is presently made only as part of *testing* H5Z-ZFP.

The `print_h5repack_farg` utility can be used to read a command-line consisting of ZFP filter parameters you wish to use and output part of the command-line needed for the `-f` argument to `h5repack`.

6.3 Examples

Note: The examples below assume H5Z-ZFP has been configured to run tests by either a preceding `make check` (if using a vanilla `gmake` build) or `make test` (if using a `cmake` build).

In the examples below, we use `h5repack` with the example data file, `mesh.h5` in the tests directory.

To use ZFP filter in *rate* mode with a rate of `4.5` bits per value, first, use the `print_h5repack_farg`:

```
% ./print_h5repack_farg zfpmode=1 rate=4.5

Print cdvals for set of ZFP compression paramaters...
    zfpmode=1  set zfp mode (1=rate,2=prec,3=acc,4=expert,5=rev)
    rate=4.5                    set rate for rate mode of filter
    acc=0               set accuracy for accuracy mode of filter
    prec=0        set precision for precision mode of zfp filter
    minbits=0          set minbits for expert mode of zfp filter
    maxbits=0          set maxbits for expert mode of zfp filter
    maxprec=0          set maxprec for expert mode of zfp filter
    minexp=0            set minexp for expert mode of zfp filter
    help=0                                     this help message

h5repack -f argument...
    -f UD=32013,0,4,1,0,0,1074921472
```

Next, cut-n-paste the `-f UD=32013,0,4,1,0,0,1074921472` in a command to `h5repack` like so:

```
env LD_LIBRARY_PATH=<path-to-dir-with-libhdf5.so>:$(LD_LIBRARY_PATH) \
    HDF5_PLUGIN_PATH=<path-to-dir-with-libh5zzfp.so> \
    $(HDF5_BIN)/h5repack -f UD=32013,0,4,1,0,0,1074921472 \
         -l Pressure,Pressure2,Pressure3:CHUNK=10x20x5 \
         -l Velocity,Velocity2,Velocity3,VelocityZ,VelocityZ2,
↪VelocityZ3:CHUNK=11x21x1x1 \
         -l VelocityX_2D:CHUNK=21x31 \
         mesh.h5 mesh_repack.h5
```

where the `-l` arguments indicate the dataset(s) to be re-packed as well as their (new) chunking.

To use ZFP filter in *accuracy* mode with an accuracy of `0.075`, first, use the `print_h5repack_farg`:

```
% ./print_h5repack_farg zfpmode=3 acc=0.075

Print cdvals for set of ZFP compression paramaters...
    zfpmode=3  set zfp mode (1=rate,2=prec,3=acc,4=expert,5=rev)
    rate=3.5                    set rate for rate mode of filter
    acc=0.075           set accuracy for accuracy mode of filter
    prec=0        set precision for precision mode of zfp filter
```

(continues on next page)

(continued from previous page)

```
    minbits=0           set minbits for expert mode of zfp filter
    maxbits=0           set maxbits for expert mode of zfp filter
    maxprec=0           set maxprec for expert mode of zfp filter
    minexp=0             set minexp for expert mode of zfp filter
    help=0                                 this help message

h5repack -f argument...
    -f UD=32013,0,4,3,0,858993459,1068708659
```

Next, cut-n-paste the `-f UD=32013,0,4,3,0,858993459,1068708659` in a command to `h5repack` like so:

```
env LD_LIBRARY_PATH=<path-to-dir-with-libhdf5.so>:$(LD_LIBRARY_PATH) \
    HDF5_PLUGIN_PATH=<path-to-dir-with-libh5zzfp.so> \
    $(HDF5_BIN)/h5repack -f UD=32013,0,4,3,0,858993459,1068708659 \
         -l Pressure,Pressure2,Pressure3:CHUNK=10x20x5 \
         -l Velocity,Velocity2,Velocity3,VelocityZ,VelocityZ2,
↪VelocityZ3:CHUNK=11x21x1x1 \
         -l VelocityX_2D:CHUNK=21x31 \
         mesh.h5 mesh_repack.h5
```

CHAPTER 7

Endian Issues

This section describes some issues related to endianness of producers and consumers of the data processed by H5Z-ZFP. This is likely less of an issue than it once was because almost all modern CPUs are little-endian.

That being said, the ZFP library writes an endian-independent stream.

There is an unavoidable inefficiency when reading ZFP compressed data on a machine with a different endianness than the writer (e.g. a *mixed* endian context). Upon reading data from storage and decompressing the read stream with ZFP, the correct endianness is returned in the result from ZFP before the buffer is handed back to HDF5 from the decompression filter. This happens regardless of reader and writer endianness incompatibility. However, the HDF5 library expects to get from H5Z-ZFP the endianness of the data as it was stored to the file on the writer machine and expects to have to byte-swap that buffer before returning to it an endian-incompatible caller.

This means that in the H5Z-ZFP plugin, we wind up having to un-byte-swap an already correct result read in a cross-endian context. That way, when HDF5 gets the data and byte-swaps it as it is expecting to, it will produce the correct final result. There is an endianness test in the Makefile and two ZFP compressed example datasets for big-endian and little-endian machines to test that cross-endian reads/writes work correctly.

Again, because most CPUs are now little-endian and because ZFP became available only after the industry mostly moved away from big-endian, it is highly unlikely that this inefficiency will be triggered.

Finally, *endian-targetting*, which is setting the file datatype for an endianness that is possibly different than the native endianness of the writer, is explicitly disallowed. For example, data may be produced on a big-endian system, but most consumers will be little-endian. Therefore, to alleviate downstream consumers from having to always byte-swap, it is desirable to byte-swap to little-endian when the data is written. However, the juxtaposition of HDF5's type conversion and filter operations in a pipeline makes this impractical for the H5Z-ZFP filter. The H5Z-ZFP filter will explicitly catch this condition, fail the compression and issue an error message.

CHAPTER 8

Tests and Examples

The tests directory contains a few simple tests of the H5Z-ZFP filter some of which also serve as decent examples.

The test client, test_write.c is compiled a couple of different ways. One target is `test_write_plugin` which demonstrates the use of this filter as a standalone plugin. The other target, `test_write_lib`, demonstrates the use of the filter as an explicitly linked library. By default, these test a simple 1D array with and without ZFP compression using either the *Generic Interface* (for plugin) or the *Properties Interface* (for library). You can use the code there as an example of using the ZFP filter either as a plugin or as a library. However, these also include some advanced usages for 4D and 6D, time-varying (e.g. *extendible*) datasets. The command `test_write_lib help` or `test_write_plugin help` will print a list of the example's options and how to use them.

8.1 Write Test Options

```
./test/test_write_lib --help
    ifile=""                                            set input filename
    ofile="test_zfp.h5"                                 set output filename

1D dataset generation arguments...
    npoints=1024             set number of points for 1D dataset
    noise=0.001         set amount of random noise in 1D dataset
    amp=17.7             set amplitude of sinusoid in 1D dataset
    chunk=256                      set chunk size for 1D dataset
    doint=0                              also do integer 1D data

ZFP compression paramaters...
    zfpmode=3        (1=rate,2=prec,3=acc,4=expert,5=reversible)
    rate=4                        set rate for rate mode of filter
    acc=0                 set accuracy for accuracy mode of filter
    prec=11       set precision for precision mode of zfp filter
    minbits=0           set minbits for expert mode of zfp filter
    maxbits=4171        set maxbits for expert mode of zfp filter
    maxprec=64          set maxprec for expert mode of zfp filter
    minexp=-1074         set minexp for expert mode of zfp filter
```

(continues on next page)

(continued from previous page)

```
Advanced cases...
    highd=0                                         run 4D case
    sixd=0          run 6D extendable case (requires ZFP>=0.5.4)
    help=0                                    this help message
```

The test normally just tests compression of 1D array of integer and double precision data of a sinusoidal array with a small amount of additive random noise. The `highd` test runs a test on a 4D dataset where two of the 4 dimensions are not correlated. This tests the plugin's ability to properly set chunking for HDF5 such that chunks span **only** correlated dimensions and have non-unity sizes in 3 or fewer dimensions. The `sixd` test runs a test on a 6D, extendible dataset representing an example of using ZFP for compression along the *time* axis.

There is a companion, test_read.c which is compiled into `test_read_plugin` and `test_read_lib` which demonstrates use of the filter reading data as a plugin or library. Also, the commands `test_read_lib help` and `test_read_plugin help` will print a list of the command line options.

To use the plugin examples, you need to tell the HDF5 library where to find the H5Z-ZFP plugin with the `HDF5_PLUGIN_PATH` environment variable. The value you pass is the path to the directory containing the plugin shared library.

Finally, there is a Fortran test example, test_rw_fortran.F90. The Fortran test writes and reads a 2D dataset. However, the Fortran test is designed to use the filter **only** as a library and not as a plugin. The reason for this is that the filter controls involve passing combinations of integer and floating point data from Fortran callers and this can be done only through the *Properties Interface*, which by its nature requires any Fortran application to have to link with an implementation of that interface. Since we need to link extra code for Fortran, we may as well also link to the filter itself alleviating the need to use the filter as a plugin. Also, if you want to use Fortran support, the HDF5 library must have, of course, been configured and built with Fortran support as well.

In addition, a number tests are performed in the Makefile which test the plugin by using some of the HDF5 tools such as `h5dump` and `h5repack`. Again, to use these tools to read data compressed with the H5Z-ZFP filter, you will need to inform the HDF5 library where to find the filter plugin. For example..

```
env HDF5_PLUGIN_PATH=<dir> h5ls test_zfp.h5
```

Where `<dir>` is the relative or absolute path to a directory containing the filter plugin shared library.