H5Z-ZFP Documentation

Release 0.6.0

H5Z-ZFP

May 28, 2019

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

1.1 Installing via Spack

The HDF5 and ZFP libraries and the H5Z-ZFP plugin are all now part of the Spack package manager. If you already have Spack installed, the easiest way to install H5Z-ZFP is to simply use the Spack command `spack install h5z-zfp`. If you do not have Spack installed, it is very easy to install.

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

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 that 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). 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 your 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

1.2 Manual Installation

If Spack is not an option for you, information on *manually* installing is provided here.

1.2.1 Prerequisites

- ZFP Library (or from Github)

- HDF5 Library
- H5Z-ZFP filter plugin

1.2.2 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.

1.2.3 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.

1.3 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`

1.4 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`.

1.5 Silo Integration

This filter 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 for H5Z-ZFP, you will see some logic there that is specific to using this plugin within Silo and dealing with ZFP as an embedded library using this struct of function pointers wrapper. Just ignore this.

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 properties 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.

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("%d cd_values= ",cd_nelmts);
    for (i = 0; i < 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)) 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.

Because of the type punning involved, the generic interface is not suitable for Fortran callers.

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);
```

(continues on next page)

(continued from previous page)

```
herr_t H5Pset_zfp_accuracy(hid_t dcpl_id, double acc);
herr_t H5Pset_zfp_expert(hid_t dcpl_id,
    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

A Fortran interface based on the properties interface, described above, has been added by Scot Breitenfeld of the HDF5 group. The code that implements the Fortran interface is in the file `H5Zzfp_props_f.F90`. An example of its use is in `test/test_rw_fortran.F90`. The properties interface is the only interface available for Fortran callers.

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 that whenever possible 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 it will pad with useless data reducing overall time and space efficiency of the results.

The degree to which this may degrade performance depends on the percentage of a chunk that is padded. Suppose we have 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 potentialy very relevant when attemping to apply ZFP to compress data long 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. 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 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 no 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 is chosoen 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 fd, 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))) ERROR(H5Screate_simple);

        /* write the data WITHOUT compression */
        if (0 > (dsid = H5Dcreate(fid, "highD_original", H5T_NATIVE_DOUBLE, sid, H5P_
↪DEFAULT, H5P_DEFAULT, H5P_DEFAULT))) ERROR(H5Dcreate);
        if (0 > H5Dwrite(dsid, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL, H5P_DEFAULT,␣
↪buf)) ERROR(H5Dwrite);
        if (0 > H5Dclose(dsid)) ERROR(H5Dclose);

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

        /* clean up from high dimensional test */
        if (0 > H5Sclose(sid)) ERROR(H5Sclose);
        if (0 > H5Pclose(cpid)) 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 patters 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.

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

3.3 Compression *Along* the *State Iteration* Dimension

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

The challenge here is to manage the data in a way that meets 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*. 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 snipit 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* will ulimately get overwritten with new timesteps.

```
1       /* Test six dimensional, time varying array...
2              ...a 3x3 tensor valued variable
3              ...over a 3D+time domain.
4              Dimension sizes are chosen to miss perfect ZFP block alignment.
5       */
6       if (sixd)
7       {
8           void *tbuf;
9           int t, fd, dims[] = {31,31,31,3,3}; /* a single time instance */
10          int ucdims[]={3,4}; /* indices of UNcorrleted dimensions in dims (tensor␣
   ↪components) */
11          hsize_t  hdims[] = {31,31,31,3,3,H5S_UNLIMITED};
12          hsize_t hchunk[] = {31,31,31,1,1,4}; /* 4 non-unity, requires >= ZFP 0.5.4 */
13          hsize_t hwrite[] = {31,31,31,3,3,4}; /* size/shape of any given H5Dwrite */
14
15          /* Setup the filter properties and create the dataset */
16          cpid = setup_filter(6, hchunk, zfpmode, rate, acc, prec, minbits, maxbits,␣
   ↪maxprec, minexp);
17
18          /* Create the time-varying, 6D dataset */
19          if (0 > (sid = H5Screate_simple(6, hwrite, hdims))) ERROR(H5Screate_simple);
20          if (0 > (dsid = H5Dcreate(fid, "6D_extendible", H5T_NATIVE_DOUBLE, sid, H5P_
   ↪DEFAULT, cpid, H5P_DEFAULT))) ERROR(H5Dcreate);
21          if (0 > H5Sclose(sid)) ERROR(H5Sclose);
22          if (0 > H5Pclose(cpid)) ERROR(H5Pclose);
23
24          /* Generate a single buffer which we'll modulate by a time-varying function
25             to represent each timestep */
26          buf = gen_random_correlated_array(TYPDBL, 5, dims, 2, ucdims);
```

(continues on next page)

(continued from previous page)

```

        /* 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;

            /* 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))) ERROR(H5Screate_simple);

            /* Get the file dataspace to use for this H5Dwrite call */
            if (0 > (fsid = H5Dget_space(dsid))) 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))␣
↪ERROR(H5Sselect_hyperslab);

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

(continues on next page)

(continued from previous page)

```
        free(tbuf);
    }
```

CHAPTER 4

Endian Issues

The ZFP library writes an endian-independent stream.

When reading ZFP compressed data on a machine with a different endian-ness than the writer, there is an unnavoidable inefficiency. Upon reading data from disk and decompressing the read stream with ZFP, the correct endian-ness 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 endian-ness incompatability. However, the HDF5 library is expecting to get from the decompression filter the endian-ness of the data as it was stored to to file (typically that of the writer machine) and expects to have to byte-swap that buffer before returning to any endian-incompatible caller. So, 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, it will produce the correct result. There is an endian-ness 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.

Finally, *endian-targetting*, that is setting the file datatype for an endian-ness that is possibly different than the native endian-ness of the writer, is currently dis-allowed with H5Z-ZFP because it is really a non-sensical operation with this filter. Since ZFP writes an endian-independent format, there is really no such thing as *endian-targetting*.

CHAPTER 5

Tests and Examples

The tests directory contains a few simple tests of the H5Z-ZFP filter.

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. 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. 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.

5.1 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.