RFC: Direct Chunk Write

Raymond Lu

This document discusses a new HDF5 C function to bypass the library's data conversion and filter pipeline and write data chunks directly to a dataset in the file. It studies the advantage for the function location in the HDF5 library and high-level library. In the end, it recommends putting the new function in the HDF5 high-level C library.

1 Introduction

A customer of the HDF Group requested a new function, which would allow an application to write pre-compressed data chunks directly to a dataset in an HDF5 file. The application will compress its data outside of the HDF5 library so that it is no longer limited by the HDF5 library's filter pipeline.

2 Motivation

The customer designs next generation X-ray pixel detectors at synchrotron light sources. It produces data at the rate of tens of gigabyte per second. The modular architecture of the detector can scale up its data stream in parallel. It maps well to the current parallel computing and storage systems. The data stream can be compressed well. The data volume can be reduced by a factor of ten or more after compression. Hence, it is crucial to compress the data before storing on disk. Any file format for storing detector data must support data compression and allow the scaling up of the number of processing nodes.

The current HDF5 library restricts compression to its filter pipeline and thus limits the performance within a single process. To overcome this weakness, the library should provide a new function to allow applications to write compressed data chunks directly to a dataset in an HDF5 file. It allows the application to compress the detector data in parallel and match the data production rate of the detector.

3 Requirements

The customer gave us the following requirements for the function.

3.1 Functionality

The new function should

- Accept converted and compressed data chunks in a memory buffer
- Provide a parameter for a chunk index that identifies the chunk's position in its dataset
- Provide a parameter to indicate which filters of the dataset's filter pipeline should be skipped when reading the chunk back from the file
- Accept a simple value 0 for the filter status to indicate that no filter should be skipped in order to simplify the application
- Be available in a release of version 1.8.11 and 1.10

3.2 Implementation

The implementation of the new function in the library should

- Use the application's original data buffer in memory to write the data chunk in the file without copying the data to any internal buffer
- Apply no modification or check to the data in the application's memory buffer
- Accept from the application data of any arbitrary non-zero size
- Allow the application to reuse the memory buffer once the library returns from the function
- Allow the writing of a partial chunk on the edge of a dataset
- Allow the overwriting of existing data chunks

4 Possible Solutions

The new function differs from the current HDF5 H5D interface functions. It takes applications' data and writes it directly to a file bypassing the HDF5 library's internals. This is the first function of this type that The HDF Group developers were tasked to implement. The current library architecture does not have a good place for adding such functions. Changes to the HDF5 library architecture are required. This section discusses three possible locations for the new function: in the HDF5 C library H5D interface, in the HDF5 high-level C library, or in the HDF5 C library but in a new library source file.

4.1 In the HDF5 C Library

The first solution proposes that the new function is added to the library among other HDF5 H5D C API functions. Below is the function prototype (`DO` stands for dataset optimization):

herr_t `H5DOwrite_chunk`(*hid_t* `dset_id`,
hid_t `dxpl_id`,
uint32_t `filter_mask`,
hsize_t *`offset`,
size_t `data_size`,
const void *`buf`)

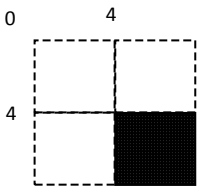
Please see *Appendix B* for the full description of this function. Basically, this function takes a pre-processed data chunk (`buf`) and its size (`data_size`) and writes to the chunk location (`offset`) in the dataset (`dset_id`).

Following is a simple example of using `H5DOwrite_chunk` to write one chunk of data, as shown in Figure 1 (a more complete example is in *Appendix B*):

```
hsize_t offset[2] = {4, 4};
uint32_t filter_mask = 0;
size_t nbytes = 40;
if(H5DOwrite_chunk(dset_id, dxpl, filter_mask, offset, nbytes, outbuf) < 0)
      goto error;
```

In this example, the dataset is 8x8 elements of `int`. Each chunk is 4x4. The offset of the first element of the chunk to be written is 4 and 4. In the diagram below (Figure 1), the shaded chunk is the data to be written.

Figure 1. Illustration of the chunk to be written in the example code above



The function is writing a pre-compressed data chunk of 40 bytes (assumed) to the dataset. The offset of the first element of the chunk is 4 and 4. The zero value of the filter mask means that all filters have been applied to the pre-processed data.

Discussion

This approach was implemented when we tested the feasibility of the function and performance. The benchmarks ran by the customer and The HDF Group developers showed an order of magnitude enhancement in I/O. The implementation met the requirements and expectation of the customer.

Integrating this function into the library does not require much work. Using this function will evoke little overhead as it deals with the low-level routines inside the library.

However, to use it appropriately, one has to understand exactly what this function does – bypassing data conversion, chunk cache, filter pipeline, etc. Since it touches the low-level library directly, it lacks the protection that most public functions have. It is more specialized compared to other general public functions. It requires the user to have special knowledge to use this function properly.

Maintenance of this function will require careful consideration since it has a different architecture and may be confusing to future maintainers of the HDF5 H5D interface.

Design

Figure 2 below shows how using `H5DOwrite_chunk` to write pre-compressed data chunks can bypass the library's data conversion and filter pipeline.

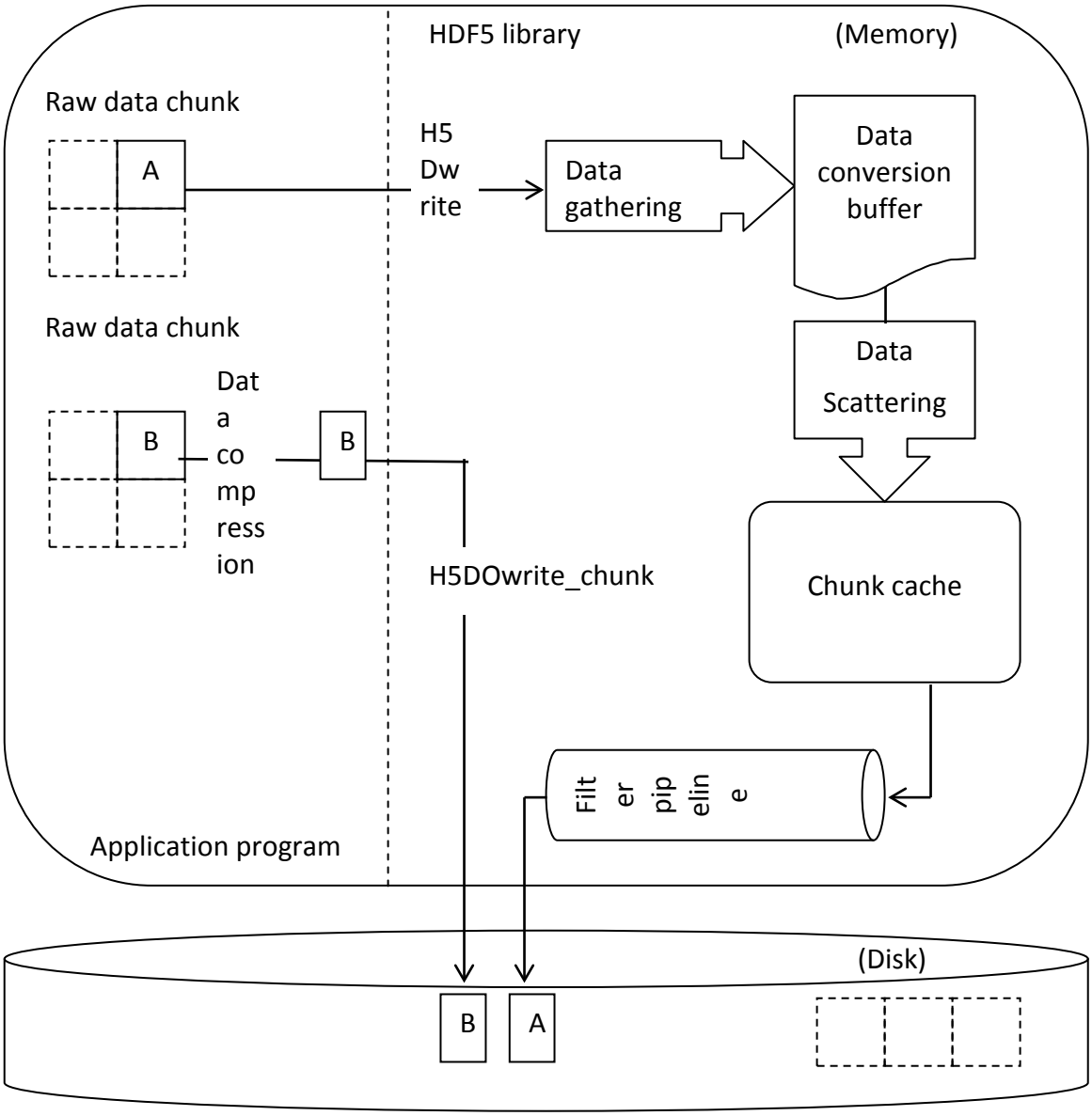
Implementation

In the internal routine `H5D__chunk_direct_write` of `H5Dchunk.c` under the library's source directory, the functionality of direct chunk write can be achieved in the following steps:

1. Initialize the chunked storage (such as B-tree indexing for chunks) through `H5D__alloc_storage` if it has not been done yet.
2. Calculate the chunk index with the chunk offset passed in as a parameter through `H5V_chunk_index`.

3. Find out the file address of the chunk to be written through `H5D__chunk_lookup`.
4. Create the chunk if it does not exist, or re-allocate the chunk if its size changes through a callback function for insertion.
5. Evict the entry for the chunk through `H5D__chunk_cache_evict` if it is in the chunk cache without flushing it to the file.
6. Write the chunk to the file through `H5F_block_write`.

Figure 2. Diagram for `H5DOwrite_chunk` in the HDF5 Library



Performance

The following table (Table 1) describes the results of performance benchmark tests run by the HDF developers. It shows that using the new function `H5DOwrite_chunk` to write pre-compressed data is much faster than using the `H5Dwrite` function to compress and write the same data with the filter pipeline. Measurements involving `H5Dwrite` include compression time in the filter pipeline. Since the data is already compressed before `H5DOwrite_chunk` is called, use of `H5DOwrite_chunk` to write compressed data avoids the performance bottleneck in the HDF5 filter pipeline.

The test was run on a Linux 2.6.18 / 64-bit Intel x86_64 machine (koala). The dataset contains 100 chunks. Only one chunk is written to the file per write call. The number of writes is 100. The time

measurement is for the entire dataset with the Unix system function `gettimeofday`. Writing the entire dataset with one write call takes almost the same amount of time as writing chunk by chunk. In order to force the system to flush the data to the file, the `O_SYNC` flag is used to open the file.

Table 1. Performance result for `H5DOwrite_chunk`

Dataset size (MB)	95.37		762.94		2288.82	
Size after compression (MB)	64.14		512.94		1538.81	
Dataset dimensionality	100x1000x250		100x2000x1000		100x2000x3000	
Chunk dimensionality	1000x250		2000x1000		2000x3000	
Datatype	4-byte integer		4-byte integer		4-byte integer	
	speed[1]	time[2]	speed	time	speed	time
`H5Dwrite` writes without compression filter	76.22	1.25	95.66	7.98	95.4	23.99
`H5DOwrite_chunk` writes uncompressed data	76.99	1.24	96.24	7.93	94.72	24.16
`H5Dwrite` writes with compression filter	**2.68**	**35.59**	**2.68**	**284.68**	**2.66**	**860.46**
`H5DOwrite_chunk` writes compressed data	**77.31**	**0.83**	**94.71**	**5.42**	**96.91**	**15.88**
Unix writes compressed data to Unix file	78.25	0.82	95.75	5.36	98.16	15.68

[1] IO speed is in MB/s.

[2] Time is in second (s).

4.2 In the HDF5 High-level C Library

The second solution proposes that the new function is put into the high-level library. The function is still called `H5DOwrite_chunk`. It is used in the same way as the first proposal (4.1).

Discussion

This approach requires additional considerations to ones discussed in section 4.1. Following the convention that all high-level functions use only the library's public functions, this function should call `H5Dwrite`. In order to call H5Dwrite the following additional steps must be taken: the parameters for direct chunk write function `H5DOwrite_chunk` will be saved in the dataset transfer property list and be transferred into `H5D__pre_write` (a new internal function). In `H5D__pre_write`, the library will retrieve the information for direct chunk write from the dataset transfer property list and handle it as a special case. This approach has a clean organization of layering. But it requires the re-evaluation of the performance achieved with 4.1, especially from the customer's side.

Design

Figure 3 below shows the design layout for `H5DOwrite_chunk` in the high-level library and the subsequent changes to internal functions in the library source code.

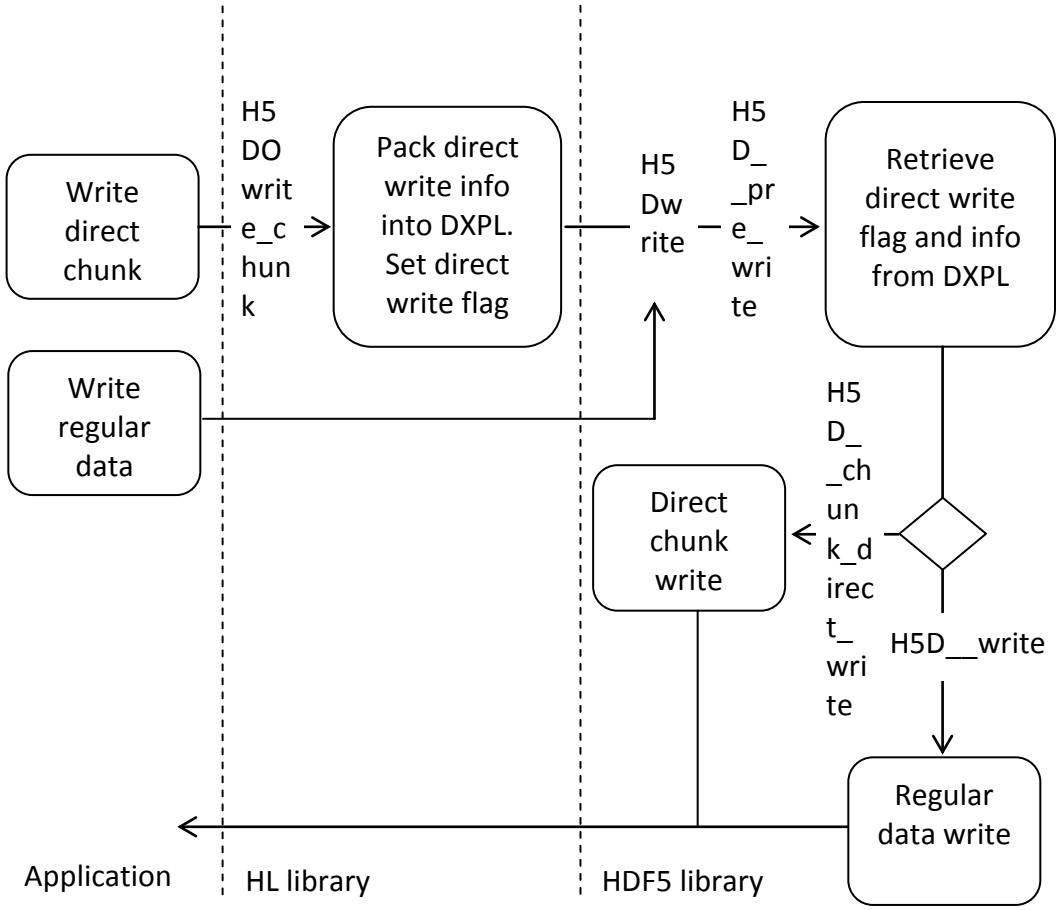
The functionality of `H5D__chunk_direct_write` has been discussed in section 4.1. It bypasses the data gathering and scattering, data conversion, filter pipeline, and chunk cache and writes the data chunk to the file directly. This diagram omits this part of the design. Please see Figure 1 in section 4.1.2 for the detail.

Implementation

The functionality of `H5DOwrite_chunk` is implemented in the following steps:

1. The `H5DOwrite_chunk` function in the high-level library sets a flag in the dataset transfer property list to indicate direct chunk write. It also saves its parameters `filter_mask`, `offset`, and `data_size` in the dataset transfer property list. Then it calls `H5Dwrite` since high-level functions can only call the library's API function.
2. Both regular writes and direct chunk writes go through `H5Dwrite`. `H5D__pre_write` (in `H5Dio.c`) is a new internal function that prepares for calling `H5D__write` or `H5D__chunk_direct_write`.
3. If the flag for direct chunk write is on, `H5D__pre_write` will retrieve the information (`filter_mask`, `offset`, and `data_size`) for direct chunk write from the dataset transfer property list and invoke `H5D__chunk_direct_write` (in `H5Dchunk.c`).
4. If it is a regular write, the direct chunk write flag is off. `H5D__pre_write` would then invoke `H5D__write` for a regular write.

Figure 3. Diagram for `H5DOwrite_chunk` in the high-level library



In the figure above, DXPL is short for the dataset transfer property list.

Performance

The performance of `H5DOwrite_chunk` in the high-level library is similar to the function's performance in the HDF5 library (see 4.1.4). The customer has confirmed that its performance is as good as they expect. The table (Table 2) below shows that using `H5DOwrite_chunk` to write pre-compressed data avoids the performance bottleneck due to compression in the filter pipeline of the library. The results are generally consistent with the results provided in Table 1.

Table 2. Performance result for `H5DOwrite_chunk` in the high-level library

Dataset size (MB)	95.37		762.94		2288.82	
Size after compression (MB)	64.14		512.94		1538.81	
Dataset dimensionality	100x1000x250		100x2000x1000		100x2000x3000	
Chunk dimensionality	1000x250		2000x1000		2000x3000	
Datatype	4-byte integer		4-byte integer		4-byte integer	
	speed[1]	time[2]	speed	time	speed	time
`H5Dwrite` writes without compression filter	77.27	1.23	97.02	7.86	91.77	24.94
`H5DOwrite_chunk` writes uncompressed data	79	1.21	95.71	7.97	89.17	25.67
`H5Dwrite` writes with compression filter	**2.68**	**35.59**	**2.67**	**285.75**	**2.67**	**857.24**
`H5DOwrite_chunk` writes compressed data	**77.19**	**0.83**	**78.56**	**6.53[3]**	**96.28**	**15.98**
Unix writes compressed data to Unix file	76.49	0.84	95	5.4	98.59	15.61

[1] IO speed is in MB/s.

[2] Time is in second (s).

[3] This number needs further investigation since it is not consistent with the similar result in Table 1.

4.3 In the HDF5 C Library but in a New Library Source File

The third solution proposes that the new function would behave in the same way as the first proposal, but that the function would be located in a separate source file.

The new source file called `H5Dopt.c`. In the future, other similar optimization functions will be located in `H5<X>opt.c`. Users would choose to include the direct chunk write feature by specifying an option on the configure line. This approach needs less work from the first round of prototype testing. The implementation is simpler than the second approach (4.2).

From the user's perspective, this approach is the same as the first approach. Users need to understand the function's behaviors – how it bypasses data conversion, chunk cache, and file pipeline.

5 Recommendation

Since the performance of the "direct chunk write" feature is not affected by where the function is located, we recommend that the second proposal be adopted. The function calls `H5Dwrite` and subsequently goes through the library's routine check and protection for public functions to prevent programming and data errors. The extra protections afforded by the high-level library make the extra work worthwhile. This approach is also applicable to other functions that will be added to the HDF5 software in the near future thus providing uniform approach for the functions that do not follow the mainstream architecture.

Since this new function `H5DOwrite_chunk` writes a data chunk directly in file, users must be careful in using it. It bypasses hyperslab selection, data conversion, and filter pipeline to write the chunk. Users must know exactly where to write the data and how to process the data before the write.

Revision History

November 14, 2012:	Version 1 circulated for comment within The HDF Group.
November 28, 2012:	Version 2 circulated for comment within The HDF Group.
November 30, 2012:	Version 3 sent to the customer.
January 8, 2013:	Version 4 circulated for comment within The HDF Group.
January 25, 2013:	Version 5 circulated for comment in the HDF Forum.

Appendix A: Background Material

In June 2012, the user provided us a requirement document called Extension to the HDF5 API and Library: Write Pre-compressed Chunk Data.

Appendix B: Reference Manual Function Entry for `H5DOwrite_chunk`

Last modified: 14 November 2012

Name: `H5DOwrite_chunk`

Signature:

herr_t `H5DOwrite_chunk`(*hid_t* `dset_id`, *hid_t* `dxpl_id`, *uint32_t* `filter_mask`, *hsize_t* * `offset`, *size_t* `data_size`, *const void* * `buf`)

Purpose:

Writes a raw data chunk from a buffer directly to a dataset.

Description:

`H5DOwrite_chunk` writes a raw data chunk from the application memory buffer `buf` directly to its logical destination `offset` in a dataset identified by `dset_id`. Typically, the data in `buf` is preprocessed in memory by a custom transformation, such as compression. The data chunk will bypass the library's internal data transfer pipeline (including filters), and will be written to the file directly.

`dxpl_id` is the identifier of a data transfer property list. (Currently, it is unused.)

`filter_mask` is a mask to keep a record of which filters are used and is saved with the data chunk in the file. The default value is zero and indicates that all enabled filters are applied. A filter is skipped if the bit corresponding to the filter position `(0<= * <32)` in the pipeline is turned on.

`offset` is an array that represents the logical position of the first element of the data chunk in the dataset's dataspace. The length of the `offset` array must equal the rank (number of dimensions) of the dataspace. The values in `offset` must not exceed the dimension limits and must fall on the boundary of data chunks.

`data_size` is the size of the raw data chunk and represents the number of bytes to be read from the buffer `buf`. If the data chunk has been pre-compressed, it should be the size of the compressed data.

`buf` is the memory buffer for the data chunk.

Parameters:

hid_t `dset_id`	IN: Identifier of the dataset to write to.
hid_t `dxpl_id`	IN: Identifier of a transfer property list for this I/O operation.
uint32_t `filter_mask`	IN: Mask for filters.
hsize_t * `offset`	IN: Logical position of the chunk's first element in the dataspace.
size_t `data_size`	IN: Size of the actual data.
const void * `buf`	IN: Buffer with data to be written to the file.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Example Usage:

Following is an example of using `H5DOwrite_chunk` to write an entire dataset by chunk:

```
#include <zlib.h>
#include <math.h>
#define DEFLATE_SIZE_ADJUST(s) (ceil(((double)(s))*1.001)+12)
             :
size_t       buf_size = CHUNK_NX*CHUNK_NY*sizeof(int);
const Bytef *z_src = (const Bytef*)(direct_buf);
Bytef       *z_dst;             /*destination buffer            */
uLongf       z_dst_nbytes = (uLongf)DEFLATE_SIZE_ADJUST(buf_size);
uLong        z_src_nbytes = (uLong)buf_size;
int          aggression = 9;     /* Compression aggression setting */
uint32_t     filter_mask = 0;
size_t       buf_size = CHUNK_NX*CHUNK_NY*sizeof(int);

/* Create the data space */
if((dataspace = H5Screate_simple(RANK, dims, maxdims)) < 0)
       goto error;

/* Create a new file */
if((file = H5Fcreate(FILE_NAME5, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT)) < 0)
       goto error;

/* Modify dataset creation properties, i.e. enable chunking and compression */
if((cparms = H5Pcreate(H5P_DATASET_CREATE)) < 0)
       goto error;

if((status = H5Pset_chunk( cparms, RANK, chunk_dims)) < 0)
       goto error;

if((status = H5Pset_deflate( cparms, aggression)) < 0)
       goto error;
```

```
/* Create a new dataset within the file using cparms creation properties */
if((dset_id = H5Dcreate2(file, DATASETNAME, H5T_NATIVE_INT, dataspace, H5P_DEFAULT,
                         cparms, H5P_DEFAULT)) < 0)
        goto error;

/* Initialize data for one chunk */
for(i = n = 0; i < CHUNK_NX; i++)
    for(j = 0; j < CHUNK_NY; j++)
        direct_buf[i][j] = n++;

/* Allocate output (compressed) buffer */
outbuf = malloc(z_dst_nbytes);
z_dst = (Bytef *)outbuf;

/* Perform compression from the source to the destination buffer */
ret = compress2(z_dst, &z_dst_nbytes, z_src, z_src_nbytes, aggression);

/* Check for various zlib errors */
if(Z_BUF_ERROR == ret) {
       fprintf(stderr, "overflow");
       goto error;
} else if(Z_MEM_ERROR == ret) {
       fprintf(stderr, "deflate memory error");
       goto error;
} else if(Z_OK != ret) {
       fprintf(stderr, "other deflate error");
       goto error;
}

/* Write the compressed chunk data repeatedly to cover all the chunks in the
 * dataset, using the direct write function.     */
for(i=0; i<NX/CHUNK_NX; i++) {
       for(j=0; j<NY/CHUNK_NY; j++) {
              status = H5DOwrite_chunk(dset_id, H5P_DEFAULT, filter_mask, offset,
                                       z_dst_nbytes, outbuf);
              offset[1] += CHUNK_NY;
```

```
            }
            offset[0] += CHUNK_NX;
            offset[1] = 0;
      }

      /* Overwrite the first chunk with uncompressed data.  Set the filter mask to
       * indicate the compression filter is skipped */
      filter_mask = 0x00000001;
      offset[0] = offset[1] = 0;
      if(H5DOwrite_chunk(dset_id, H5P_DEFAULT, filter_mask, offset, buf_size, direct_buf)
< 0)
            goto error;

      /* Read the entire dataset back for data verification converting ints to longs*/
      if(H5Dread(dataset, H5T_NATIVE_LONG, H5S_ALL, H5S_ALL, H5P_DEFAULT,
                  outbuf_long) < 0)
            goto error;

      /* Data verification here */
                  :
                  :
```