

## **Profiling Write Performance of HDF5**

J. Hill, L. Smith, A. Trew

EPCC, University of Edinburgh, JCMB, Kings Buildings, Edinburgh EH9 3JZ

### **Abstract**

The performance of the portable, binary data format, HDF5 has been explored on HPCx, the national HPC service using OCCAM test data. Results using chunked, compressed data, written in hyperslabs, show that a 64KB chunk size is optimal. Compression is the main degrading factor on performance. If chunking is used then the user must ensure that the chunk cache size is set to be at least equal to the size of a single chunk. Other HDF5 features were tested and resulted in a slight decrease in performance or had no discernable effect. In general the default values of the parameters in HDF5 will give good performance for most users. The greatest increase in performance occurred when both HDF5 and the compression libraries were compiled by the user rather than using the default implementation available on the HPCx system. This resulted in an almost five-fold increase in performance.

**This is a Technical Report from the HPCx Consortium.**

Report available from <http://www.hpcx.ac.uk/research/publications/HPCxTR0601.pdf>

**© UoE HPCx Ltd 2005**

Neither UoE HPCx Ltd nor its members separately accept any responsibility for loss or damage arising from the use of information contained in any of their reports or in any communication about their tests or investigations.

---

<b>1</b>	<b><i>Introduction</i></b>	<b>3</b>
<b>2</b>	<b><i>Methods</i></b>	<b>3</b>
<b>3</b>	<b><i>HDF5</i></b>	<b>4</b>
<b>3.1</b>	<b>HDF5 Features</b>	<b>5</b>
3.1.1	Chunking	5
3.1.2	Hyperslabs	6
3.1.3	Compression and Shuffling	7
3.1.4	Parallel HDF5	8
3.1.5	Writing Files with HDF5	8
3.1.6	Property List Options	9
<b>4</b>	<b><i>OCCAM and HDF5</i></b>	<b>10</b>
<b>4.1</b>	<b>Data Structure in OCCAM</b>	<b>10</b>
<b>5</b>	<b><i>Results</i></b>	<b>10</b>
<b>5.1</b>	<b>Platform</b>	<b>10</b>
<b>5.2</b>	<b>Profiling</b>	<b>11</b>
<b>5.3</b>	<b>Benchmarking Tests</b>	<b>11</b>
<b>5.4</b>	<b>Chunk Size Tests</b>	<b>12</b>
<b>5.5</b>	<b>Chunk Cache Tests</b>	<b>14</b>
5.5.1	Cache Levels	14
5.5.2	Cache Size	14
5.5.3	Cache Size with Small Chunks	15
5.5.4	Pre-Emptive Policy	16
<b>5.6</b>	<b>Other Buffer Tests</b>	<b>17</b>
5.6.1	Sieve Buffer Size	17
5.6.2	Buffer Size	17
<b>5.7</b>	<b>Compression</b>	<b>18</b>
5.7.1	gzip Compression Level	18
5.7.2	szip Compression	19
5.7.3	gzip Compression	19
<b>5.8</b>	<b>Other Tests</b>	<b>20</b>
5.8.1	Data Block Size	20
5.8.2	Meta Block Size	21
5.8.3	Shuffle	21
<b>5.9</b>	<b>Parallel vs. Serial HDF5</b>	<b>21</b>
<b>5.10</b>	<b>64 vs. 32 bit HDF5</b>	<b>22</b>
<b>6</b>	<b><i>Conclusions</i></b>	<b>22</b>
<b>7</b>	<b><i>Acknowledgements</i></b>	<b>23</b>
<b>8</b>	<b><i>References</i></b>	<b>23</b>

## 1 Introduction

OCCAM [1], the global ocean modeling software, outputs large volumes of data after each successful run. The data is output in binary format, but is then converted to a more portable format, HDF5 [2], for analysis and archiving. The conversion is done on a single processor using an OCCAM developed program: `rest2hdf5`, which utilises the HDF5 library. Currently a twelve hour OCCAM run produces four sets of five files. Each set takes 40 minutes to convert, so a twelve hour run is extended by a further 22% to archive the data. Although it is not strictly necessary to wait for the conversion process to complete before (re-)starting the next OCCAM run, analysis of the HDF5 file provide useful checks on the data.

The total size of the OCCAM output files for a 12 hour run is currently around 80GB of data. The data has been compressed by not recording any data that would be on land. The HDF5 conversion then compresses this further by around 50%, giving a final archive file of around 40GB.

For the purposes of this project a small Fortran program has been developed which emulates the behaviour of `rest2hdf5`. This should make profiling and any necessary code changes easier to implement. The test code outputs a single HDF5 file which is around 1.9GB in size. The data stored is a simulation of OCCAM data.

This study aims to reduce the time spent converting the output files and provide general optimisations for users utilising the HDF5 library on HPCx. This technical report details the optimisations used and their effect on the run time of the conversion code.

This report covers the methods used in this study before detailing some of the features of HDF5 and explains some of the more common library calls. It then shows the layout of the data in the OCCAM test code, before detailing the results of the test carried out on HPCx. Finally recommendations are made for users of HPCx who wish to include HDF5 in any software.

## 2 Methods

The methods used to optimise performance will be to alter the various cache, buffer and other parameters available in HDF5. Another option of decreasing run-time is to parallelise the conversion code as HDF5 is capable of writing file in parallel. However, this may not be a viable option as compression cannot be used while writing parallel files. The file may be compressed afterwards, but this requires additional, cumbersome and inconvenient steps when both writing and reading files.

The Fortran test code<sup>1</sup> simulates the saving of OCCAM data in HDF5 format. The software generates data and then saves this data in HDF5 format. The software can save 1 degree, 1/4 degree or 1/12 degree size datasets. In all cases the data is stored in an identical fashion, only the extent of the data changes in the X and Y directions (Z

---

<sup>1</sup> Created by Andrew Coward, National Oceanography Centre, Southampton.

remains constant).

The software has had timing routines added that will cumulatively add the time taken to create and write the HDF5 file only. The timing will be done with the `MPI_Wtime()` subroutine. The HPCx system version of HDF5 library is currently compiled with MPI.

The Fortran program supplied by the OCCAM team was first converted to a Fortran subroutine. Several “wrapper” programs were then written, which utilised the OCCAM subroutine. Each program tested a different aspect of performance enhancement.

The following experiments were carried out:

1. Profile the code to ascertain any coding bottlenecks.
2. Experiment with various buffer, cache and other settings, measuring their effect on the time taken to write the HDF5 file:
  - a. Experiment with the chunk size and chunk cache (including the number of cache levels),
  - b. Implement a sieve buffer and experiment with the size of this buffer,
  - c. Experiment with the small data block size,
  - d. Experiment with “shuffling” the data,
  - e. Experiment with the meta data block size,
  - f. Implement `szip` compression instead of `gzip`,
  - g. Experiment with the `gzip` algorithm parameters,
  - h. Experiment with the data transfer buffer.
3. Investigate the effect of using a 32-bit serial, 64-bit serial or 64-bit parallel HDF5 library
4. Investigate performance on both the Phase2 and Phase2a HPCx systems (see section 5.1).

### 3 HDF5

HDF5 is a binary file format designed to be portable without sacrificing performance [2]. HDF5 files contain two parts, binary raw data and binary meta-data. The meta-data stores the arrangement of the raw data and the architecture, endian type, size of singular datatypes, etc. This allows the HDF5 file to be written on one machine and read on any other machine with the HDF5 library. In addition, the file can be targeted to a particular architecture, for example you can set the HDF5 file to be written such that it will load rapidly on your desktop PC for repeated analysis, rather than the 64-bit supercomputer on which it was created. HDF5 also allows the writing and reading of files in parallel.

The layout of HDF5 files is somewhat like the directory structure on a Unix system (Figure 1). The file has a root directory into which various groups (equivalent to folders) can be placed. Each group can be linked to several datasets which contain a rectangular array of datatypes, which in turn can be a basic or compound (like a C struct) type. In addition, further meta-data may be stored to describe the file [2].

The performance of HDF5 has been shown to be faster than other rival portable

binary filetypes [3] and sometimes even quicker than writing the file to native binary [3]. The file size is only a fraction of a percent larger than the equivalent native binary file [4].

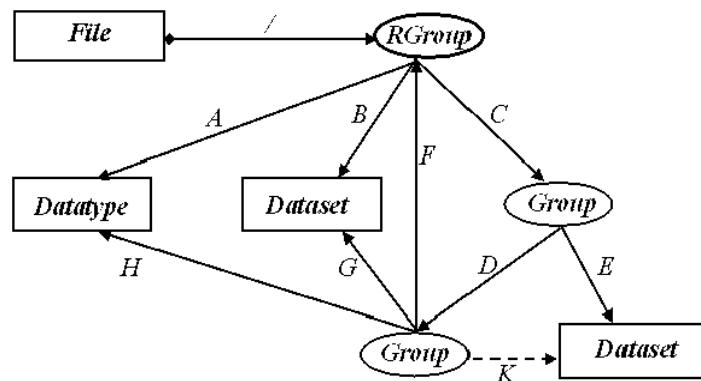


Figure 1: The organisation in a typical HDF5 file. The root of the file, “/”, is a group (RGroup) which can be linked to other groups, datasets or datatypes. The path to any element is given as a Unix-like path. For example, the path to the datatype can be /A or /C/D/H. From [1]

### 3.1 HDF5 Features

HDF5 has many other features, many of which are not mentioned in this document but can be found in [2]. Below is a summary of the features in the HDF5 library that are pertinent to this project.

#### 3.1.1 Chunking

Chunking refers to a storage layout where the dataset is partitioned into a number of fixed-size, multi-dimensional chunks [5]. The HDF5 library treats chunks as atomic objects, that is, I/O operations are carried out with complete chunks. Smaller chunks will therefore result in more I/O operations to fetch the same amount of data, but a large chunk size will consume more memory and may overwhelm the cache. Chunks are read in via a filter pipeline after which any bytes within the chunk can be altered. Writing of the chunk then occurs, again, through the filter pipeline (Figure 2).

Chunks allow portions of the dataset (which can be a whole data array or a hyperslab of the data) to be read or written at any one time, which maybe more efficient than reading in the whole dataset. If a chunk covers a portion of the dataset that does not have any bytes altered, then it is not read or written in at all. The filter pipeline allows the use of compression and other filters to be applied to the data on reading or writing.

All chunking operations can also use a cache in order to improve I/O performance. The cache size and pre-emptive policy can all be controlled by library calls. The default size of the chunk cache is 1MB.

Smaller chunks mean that more I/O operations have to be performed, but they are quicker to read or write. Smaller chunks result in a large file overhead and more I/O operations as the correct chunk is searched for. In addition, the meta-data containing the chunk properties must also be loaded through the meta-data cache. This then competes with the meta-data from the dataset itself, describing the dataset properties,

further reducing performance. However, using larger chunks requires a larger cache size, increasing memory requirements.

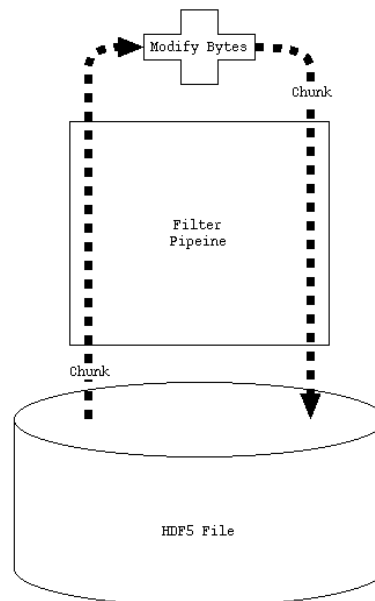


Figure 2: The HDF5 library can read in chunks of data through a filter pipeline, alter any bytes and save the chunk again. This is more efficient than reading in the whole dataset. From [4]

### 3.1.2 Hyperslabs

Hyperslabs are a way of defining a data access pattern from a dataset to memory. Hyperslabs can access and write data in numerous ways and are not limited to a one-to-one mapping of data to the same memory co-ordinates. The data can be moved to a different location in the dataset (Figure 3), accessed in a regular pattern (Figure 4) or as individual points (Figure 5). In addition hyperslabs can be overlapped either in the file or in memory to produce irregular patterns of data access (Figure 6).

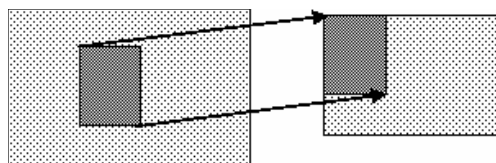


Figure 3: A one-to-one mapping of a hyperslab in an HDF5 file to a section in memory of the same size and shape

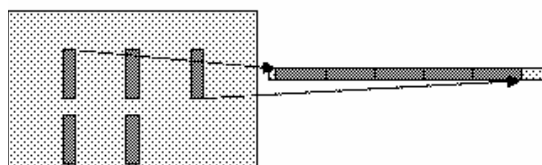


Figure 4: A regular access pattern in the HDF5 file can be transformed into a 1-D array with an offset in memory.

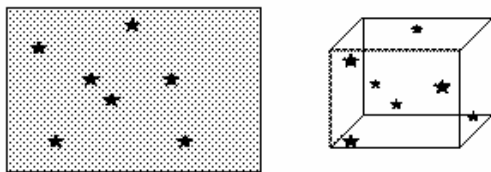


Figure 5: Single point selection can be used to transfer data from a 2-D array into 3-D space.

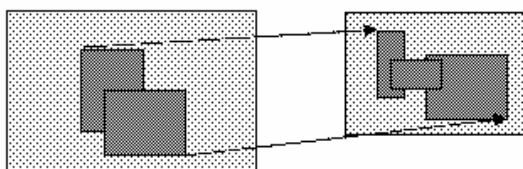


Figure 6: Hyperslabs can be combined to provide irregular data mapping. The number of elements must be the same, but the size, shape and number of hyperslabs can differ.

Reading and writing hyperslabs can be carried out with a buffer in order to improve I/O performance. The buffer should be large enough to hold at least one hyperslab in memory. The default buffer size is 1MB. However, the hyperslab buffer is to be depreciated and is liable to be removed from future versions of the HDF5 library.

### 3.1.3 Compression and Shuffling

The HDF5 library also enables the application developer to compress the binary data “on-the-fly”. The library has `gzip` compression built in, but other compression algorithms can be employed via a filter.

Shuffling is a mechanism of re-ordering the bytes in a dataset. As most scientific data has locality, i.e. the numbers have a relationship of some sort to their neighbours, re-ordering the bytes can produce a continuous stream of the same bytes. This is very advantageous to compression algorithms and hence may allow a higher level of compression when using shuffled data [6]. For example:

If we have five 32-bit unsigned integers:  
1, 23, 43, 56, 35

The hexadecimal form of these numbers is:

0x01, 0x17, 0x2B, 0x38, 0x23

On big-endian machine, these numbers are stored in memory as follows:

```
0x00 0x00 0x00 0x01 0x00 0x00 0x00 0x17 0x00 0x00 0x00 0x2B
    0x00 0x00 0x00 0x38 0x00 0x00 0x00 0x23
```

The shuffling algorithm re-arranges the byte order of these numbers, putting the first byte of every number in the first chunk and then the second byte of every number and so on.

After shuffling the data stream, in memory these numbers are stored as follows:

```
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
    0x00 0x00 0x00 0x01 0x17 0x2B 0x38 0x23
```

There are now 15 continuous zeroes in the second data stream and all non-zero numbers are at the end. This re-ordering of the bytes can enable higher compression ratios in the `gzip` and `bzip2` compression algorithms [6].

### 3.1.4 Parallel HDF5

The parallel implementation of HDF5 allows the reading and writing of HDF5 files in parallel environments, which are completely compatible with serial HDF5 files. Writing in parallel requires the use of MPI. Once a parallel file is opened in HDF5 the following can be achieved:

- All parts of the file can be accessed by all processes,
- All objects in the file can be accessed by all processes,
- Multiple processes can write to the same dataset,
- Each process can write to an individual dataset.

The only major disadvantage to using parallel I/O is that compression cannot be done using the HDF5 library. It must be done independently, after the HDF5 file has been written to disk.

### 3.1.5 Writing Files with HDF5

The first step before using HDF5 is to initialise the library. Once done, the individual properties for the file can then be set, before the list is passed to the “create” or “open” statements. There are several properties that control various aspects of the I/O.

#### ***Meta data block allocation size***

Meta data typically exists as very small chunks of data; storing meta data elements in a file without blocking them can result in hundreds or thousands of very small data elements in the file. This can result in a highly fragmented file and seriously impede I/O. By blocking meta data elements, these small elements can be grouped in larger sets, thus alleviating both problems. `H5Pset_meta_block_size` sets the minimum size in bytes of meta data block allocations.

#### ***Meta data cache***

Meta data and raw data I/O speed are often governed by the size and frequency of disk reads and writes. In many cases, the speed can be substantially improved by the use of an appropriate cache. `H5Pset_cache` sets the minimum cache size for both meta data and raw data and a pre-emption value for raw data chunks. This is currently the only method of buffering or caching used by the OCCAM HDF5 library.

#### ***Data sieve buffer size***

Data sieve buffering is used by certain file drivers to speed data I/O, most commonly when working with dataset hyperslabs. For example, using a buffer large enough to hold several pieces of a dataset as it is read in for hyperslab selections will boost performance noticeably. `H5Pset_sieve_buf_size` sets the maximum size in bytes of the data sieve buffer.

### 3.1.6 Property List Options

The property list, which controls the writing of the HDF5 file, is changed using various library calls. This list is then passed to the file creation function. For the purposes of this project the following properties may be useful. Only one of these properties is currently used by the OCCAM code; `H5Pset_cache`.

**H5Pset\_buffer** – The default size is 1MB. If the data is larger than this, HDF5 will use strip mining techniques. Increasing the buffer size to match the dataset should increase performance.

**H5Pset\_cache** – set up the cache for raw and meta-data. The default values on HPCx are 10330 for the number of meta-data elements, 521 for the number of raw data chunks, 1MB for the raw data cache in bytes and the pre-emptive policy is set to 0.75. These values are changed by the OCCAM HDF5 library. The raw data cache is increased to 8.7MB and the pre-emptive policy is set to 1.0. Both of these changes should result in increased performance [5]

**H5Pset\_hypercache** – sets the cache size for hyperslab writing. A hyperslab is a section of the whole dataset and can be a simple hyper-rectangle, a set of points, a contiguous strip made from discontinuous strips or an amalgamation of more than one hyperslab. This buffer is deprecated from HDF5, so will not be used in this study.

**H5Pset\_meta\_block\_size** – controls how big an aggregate has to be before writing to disk. Increasing the size can help reduce the number of small I/O operations, hence increasing performance.

**H5Pset\_shuffle** – shuffling the data will affect I/O performance but how it affects performance depends heavily on the data. Shuffling moves the bits around inside the data and in doing so increases CPU time for writing a file. However, by doing this the compression ratio of the data may be increased, resulting in a smaller file, which may write more quickly. If compression is not increased, then performance may be degraded by using the shuffle filter.

**H5Pset\_sieve\_buf\_size** – sets the size of the sieve buffer which controls the block size of raw data I/O. Default is 64KB. Increasing it may increase performance.

**H5Pset\_small\_data\_block\_size** – reserves some space for writing small datasets to prevent frequent I/O accesses. HDF5 assesses whether using this buffer will increase performance, but increasing it will allow HDF5 to use it for more writes than the default of 2KB.

**H5Pset\_szip** – sets up the `szip` algorithm to compress chunked data. Parameters are the mask and pixel size.

A further group of properties set when preparing the data to be written to disk rather than when creating the file, are the chunk sizes, and altering these sizes may affect performance.

## 4 OCCAM and HDF5

The current method of writing files is to create a hyperslab and write this to disk using chunks. The size of the chunks is set at 360, 8 and 11 elements in size for the X, Y and Z directions respectively. This equates to around 124KB. The chunks are compressed while writing with a compression level of one. The size of the chunk cache is changed from the default 1MB to 8.7MB, which is enough to accommodate around 72 chunks.

### 4.1 Data Structure in OCCAM

Each hyperslab occupies several lines of latitude from the OCCAM model. The  $1/12^{\text{th}}$  degree model, with the hyperslab dimensions shown in Figure 7, requires 216 whole hyperslabs to be written to disk. The software also uses chunking as described in section 3.1.1. A chunk is currently set to be much smaller than a hyperslab in the Z and X dimensions, but to be of equal size in the Y dimension (Figure 7).

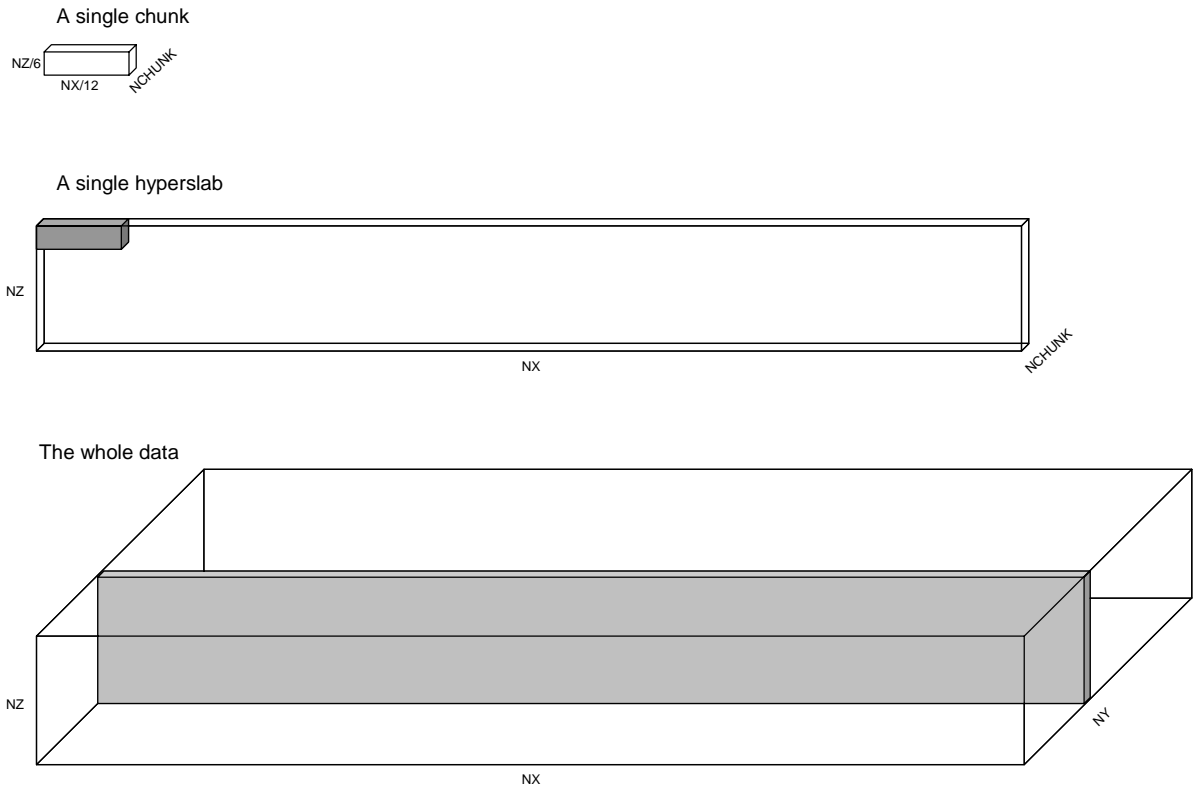


Figure 7: Data structures employed by the OCCAM conversion program. Each hyperslab (middle) is written in chunks (top). Each hyperslab is a slice of the whole data set.

## 5 Results

### 5.1 Platform

The majority of the experiments were carried out on the HPCx Phase 2 machine. The Phase 2 system of HPCx consists of 50 IBM pSeries 690+ Regatta nodes, each containing 32 1.7 GHz POWER4 processors (a total of 1600 processors). The peak computational power of the HPCx system is 10.8 TeraFlop/s, or up to at least 6

TeraFlop/s sustained. Each chip contains two processors, together with the Level 1 (L1) and Level 2 (L2) cache. On this system, each processor has its own L1 instruction cache of 64KB and L1 data cache of 32KB integrated onto one chip. The size of the L2 cache on board the chip (instructions and data) is 1.5MB, which is shared between the two processors. Four chips (8 processors) are integrated into a multi-chip module (MCM) and four MCMs (32 processors) comprise one frame. Each MCM is configured with 128MB of L3 cache and 8GB of main memory. The total L3 cache of 512MB per frame and the total main memory of 32GB per frame are shared between the 32 processors of the frame. For these tests a single processor was used, but the whole frame (32 processors) was reserved to minimise the risk of conflicts with other users when writing to the file. The system has 2 x 32-way LPARs acting as VSD servers. Access to the file infrastructure is shared between all users, which means if several users try to write a file simultaneously, they share the network bandwidth to the I/O servers. The code was compiled with moderate optimisations specified by the `-q64`, `-qarch=pwr4` and `-O3` flags.

Some further tests were carried out on the HPCx Phase 2a machine. This system utilises 1.5GHz POWER5 processors (a total of 1536). The peak computational power is 9.2 TeraFlop/s. In most respects the machine is very similar to Phase 2, but the caches on the processors have been changed. The L2 cache is now 1.9MB. The L3 cache has been moved so that it is more tightly coupled to the processor, reducing L2 cache misses. The configuration of the machine is now 2 MCMs (16 processors) per frame. The amount of main memory is the same at 32GB per LPAR, but this is used by half the number of processors. As with Phase 2, any experiments using the parallel version of HDF5 were done on a single LPAR (16 processors) to minimise conflicts. The I/O system is nearly identical to that of Phase 2, but 6 x 16-ways LPARS are used as VSD servers. The same constraints on bandwidth apply to Phase 2a also.

## 5.2 Profiling

The results from profiling the code using `gprof` show that the main bottleneck in execution time is the `gzip` library, with over 43% of the total time spent in the compression functions. Removing the compression flag resulted in a huge speed up, confirming this observation. The two profile outputs are shown in Appendix A.

## 5.3 Benchmarking Tests

All the following 14 investigations were carried out on Phase 2 of HPCx (Table 1), each testing a different HDF5 feature. Some of the runs were repeated on Phase 2a. They showed similar characteristics, but Phase 2a was around 20-25% faster. An example result is shown in Appendix B. It is also important to note that the HDF5 library is version 1.6.2 on Phase 2 and 1.6.4 on Phase 2a. Comparisons between the two versions on Phase 2a showed no significant change in performance. Many of the tests were carried out over a range of values of the parameter being investigated. Each test was repeated at least five times to average out any possible variances in I/O performance. The final results shown are the average of all five results.

All test results are compared to the “standard” run which comprises the following parameters:

- Chunk size: set to  $11 * 360 * 8$  floating point numbers or 123.75KB of data;
- Chunk cache size: set to 8.7MB;

- `gzip` compression: set to level 1;
- All other parameters are set to the HDF5 default values.

The tests results are detailed in the next few sections and are ordered as in Table 1. Error bars on Figures 8 to 21 were calculated using the standard error.

Table 1: Details of the experiments carried out.

Test	Description
<b>Chunk Size:</b>	
Chunk size	Vary the size of the chunk between 14KB and 1.5MB.
1MB sieve buffer and change chunk size	The same as the chunk size test but implement a 1MB sieve buffer. The default sieve buffer is 64KB, although it cannot be used on all file drivers.
Change chunk size, no compression	Same as above, but with no compression.
<b>Chunk Cache:</b>	
Cache levels	Vary the number of cache levels in the chunk cache between 0 and 5000. Default is 521 on HPCx.
Cache size	Vary the size of the chunk cache between 0 and 18MB. The default is 1MB on HPCx. OCCAM code had 8.7MB.
Cache size with small chunks	Using the optimum chunk size of <64KB (see above test) and decrease the cache size from 8.7MB to 0MB.
Pre-emptive Policy Test	Vary the pre-emptive policy on the chunk cache between 0 and 1.0. The default is 0.75 and OCCAM uses a value of 1.0.
<b>Other Buffers:</b>	
Sieve buffer size	Vary the size of the sieve buffer between 51KB and 8.7MB.
Buffer size	Vary the type conversion buffer size between 1KB and 18MB.
<b>Compression:</b>	
<code>szip</code> compression	Implement the <code>szip</code> compression, rather than <code>gzip</code> . The pixel size and mask mode are changed.
<code>gzip</code> compression	Vary the <code>gzip</code> compression level between 0 and 9.
<b>Other Tests:</b>	
Data block size	Vary the small data block size between 0 and 128KB. Default is 2KB in HDF5.
Meta block size	Vary the size of the meta block size between 0 and 128KB. The default is 2KB in HDF5
Shuffle	Implement a shuffle filter on the data

## 5.4 Chunk Size Tests

Three tests were carried out in total. The first test changed the chunk size along the X and Z directions. The second test increased the sieve buffer size, but otherwise the experiment was the same. The third test was identical to the second, except the compression algorithm (`gzip`) was turned off. Chunk size ranged from 14KB to 1.5MB, with the standard size being 126KB. The size was changed by altering the X

and Z dimensions only.

Results show a chunk size of 64KB is optimal on HPCx (Figure 8). The reason for this is as yet unclear. HDF5 does have a sieve buffer, which has a default size of 64KB, but repeating the experiment with a 1MB sieve buffer produces similar results (Figure 9).

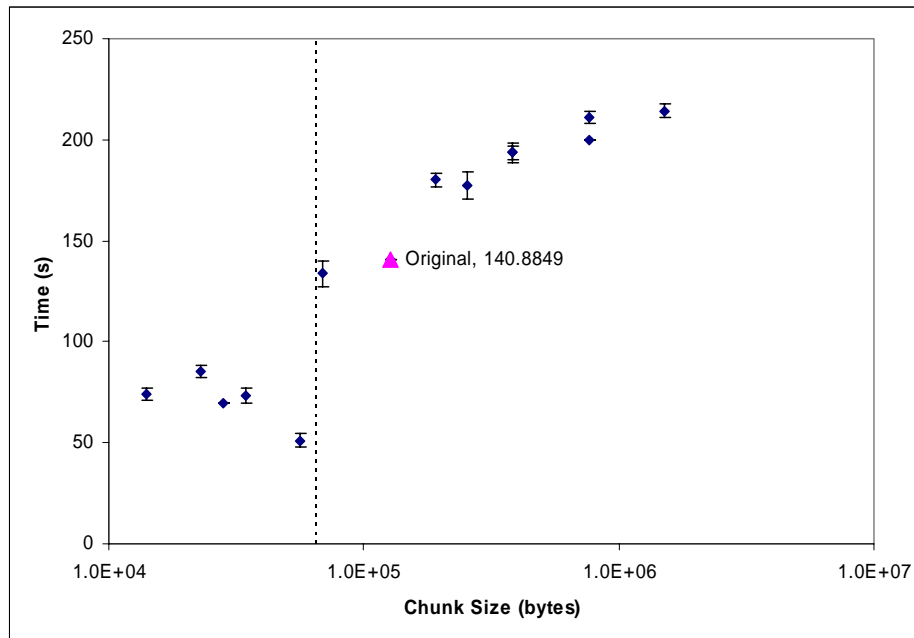


Figure 8: Changing the chunk size resulted in a near 300% increase in performance when the chunks were 64KB (dotted vertical line) in size or smaller. 64KB appears to be the optimum chunk size on HPCx. Note the logarithmic scale on Chunk Size axis. Results presented are for Phase 2.

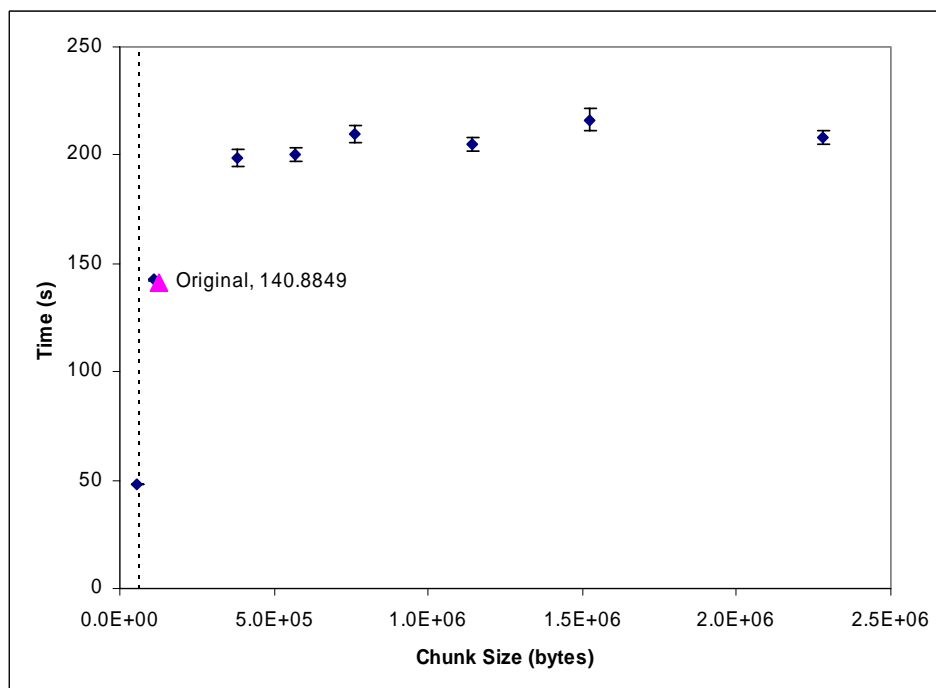


Figure 9: Increasing the sieve buffer, which has a default size of 64KB, to 1MB shows no effect. The optimum chunk size remains 64KB (dotted line). Results presented are for Phase 2.

However, as noted by the profiles, the compression algorithm appears to be the bottleneck to performance. Repeating the previous test with no compression does not show a 64KB optimum chunk size and, in fact, all chunks write quickly (Figure 10). The optimum size with no compression is just over 1MB.

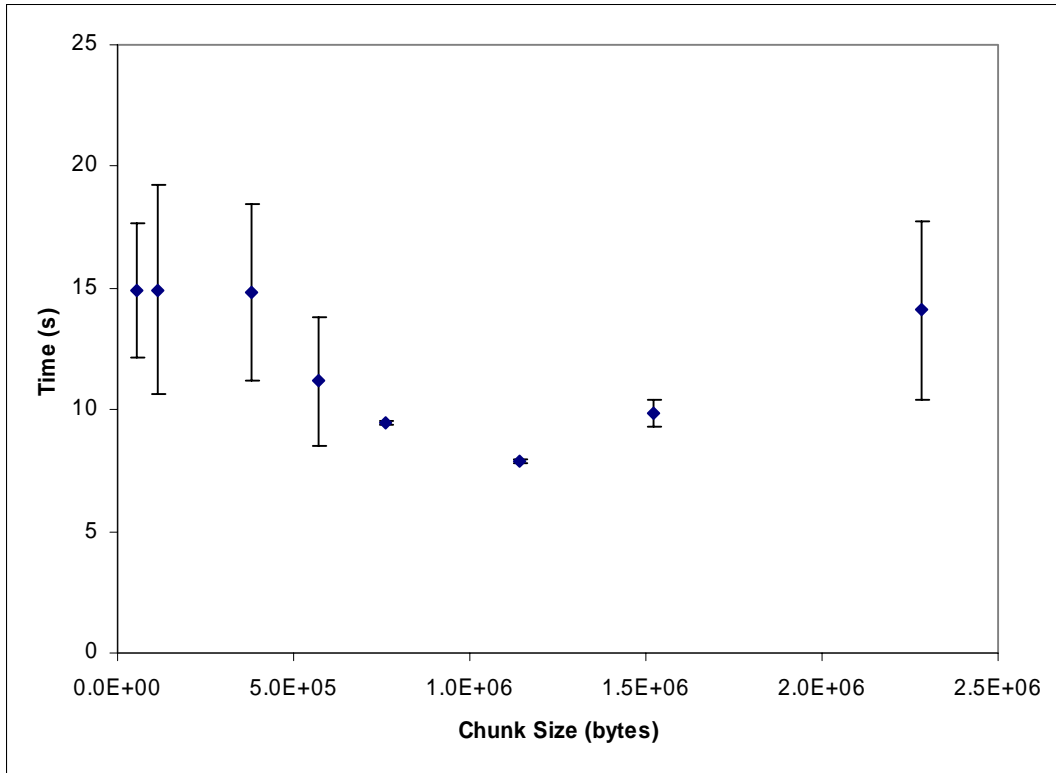


Figure 10: The quickest write times without compression occur with a chunk size of just over 1MB, although all size write quickly. Some runs took over 200 seconds on this test, but when re-run their times were similar to the other experiments. The original configuration is not shown here, but took 140.8849 seconds to complete. Results presented are for Phase 2.

## 5.5 Chunk Cache Tests

### 5.5.1 Cache Levels

The chunk cache has a number of levels within it with the default number of levels set at 521. The HDF5 manual states that the size of the chunk cache depends on both the size of the raw data cache and the number of cache levels. The actual cache size is either the raw data cache size in bytes or the number of cache levels multiplied by the chunk size; whichever is smaller. However, increasing the number of cache levels has been reported to increase read performance, although the effect on write performance was minimal and the study was carried out on an older version of the HDF5 library [7]. The size of the raw data cache was left at 8.7MB and the levels changed from 0 to 5000. The chunk has a size of 123.75KB, so the raw data cache size is equivalent to 72 chunks. The only effect seen is when the number of cache levels is set to zero – effectively the same as setting the chunk cache to zero (Figure 11).

### 5.5.2 Cache Size

The second test involving the chunk cache was to changes the size of the cache from 0 to 18MB (enough for two hyperslabs). The number of levels was left at the

default of 521. The only effect on performance occurs when the chunk cache is smaller than the size of an individual chunk (Figure 12). Increasing the cache size to well beyond that of a single chunk has no effect on performance and in fact may degrade performance slightly. The optimum chunk size appears to be around 1MB, which is the default value.

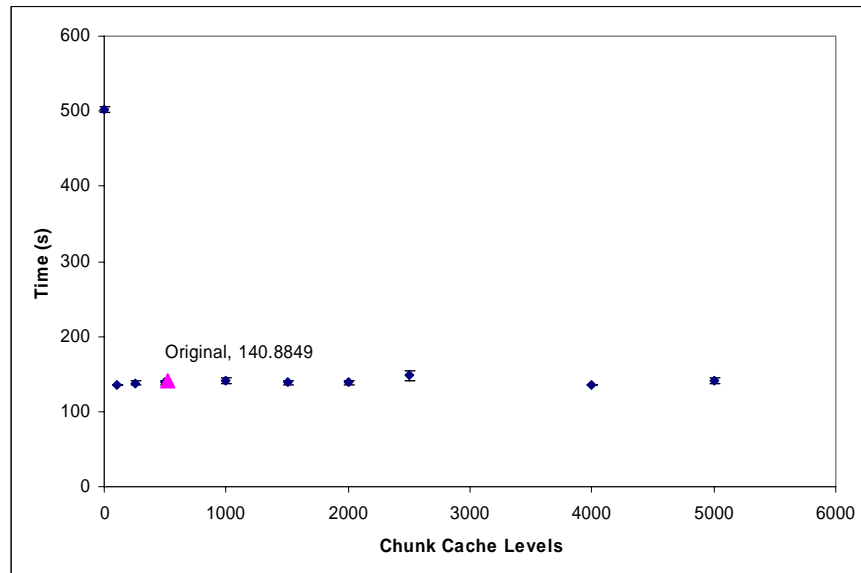


Figure 11: Changing the number of levels in the chunk cache has no effect, except when 0 levels are used, effectively the same as setting the cache size to 0B, which resulted in a massive decrease in performance. Results presented are for Phase 2.

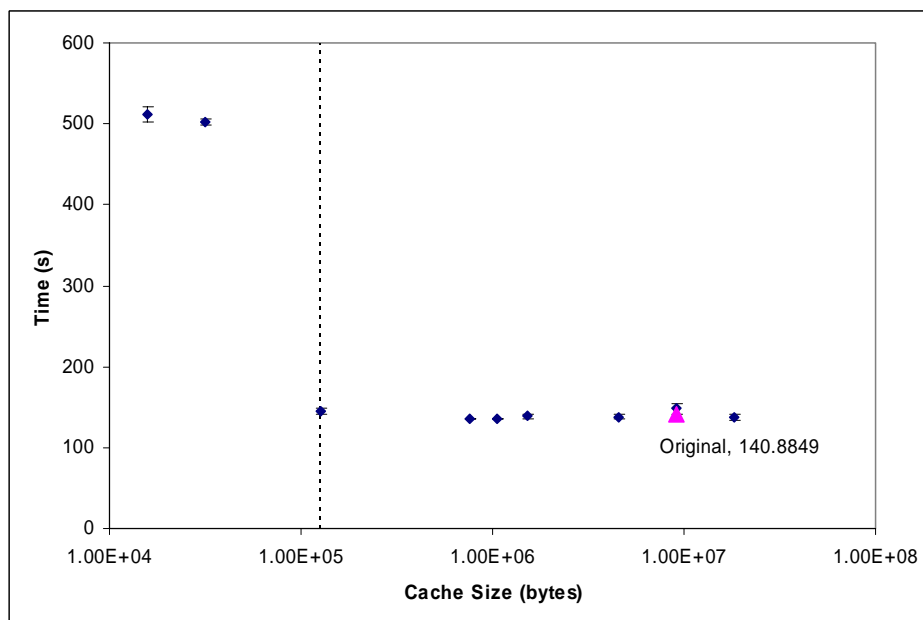


Figure 12: Decreasing the chunk size below the size of a single chunk has a marked effect on write times. However, increasing the chunk cache beyond the size of a single chunk does not increase performance. The dotted line shows the size of a single chunk. Results presented are for Phase 2.

### 5.5.3 Cache Size with Small Chunks

Repeating the test in section 5.5.2 with the optimal 64KB chunks shows similar results (Figure 13). Again, the size of the chunk cache does not have any marked

effect unless the cache is smaller than a single chunk.

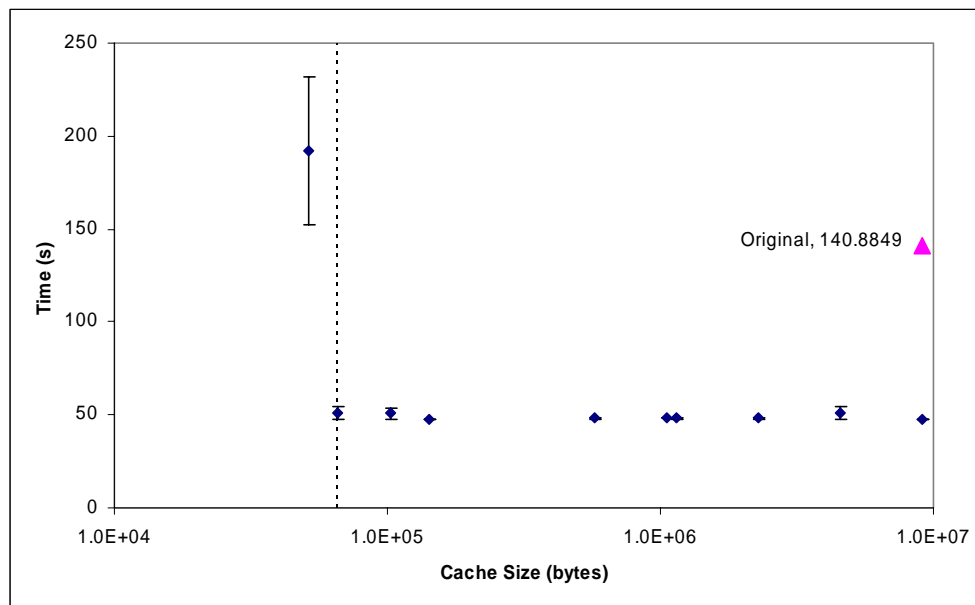


Figure 13: Decreasing the chunk size to 64KB and changing the size of the cache shows similar results to those in Figure 12. The dotted line shows the chunk size of 64KB. Results presented are for Phase 2.

### 5.5.4 Pre-Emptive Policy

The pre-emptive policy on the chunk cache is a weighting factor used by HDF5 to try and “guess” the next chunk that will be written [8]. In the case of the test OCCAM data, the chunks are written sequentially, so this parameter has no effect (Figure 14).

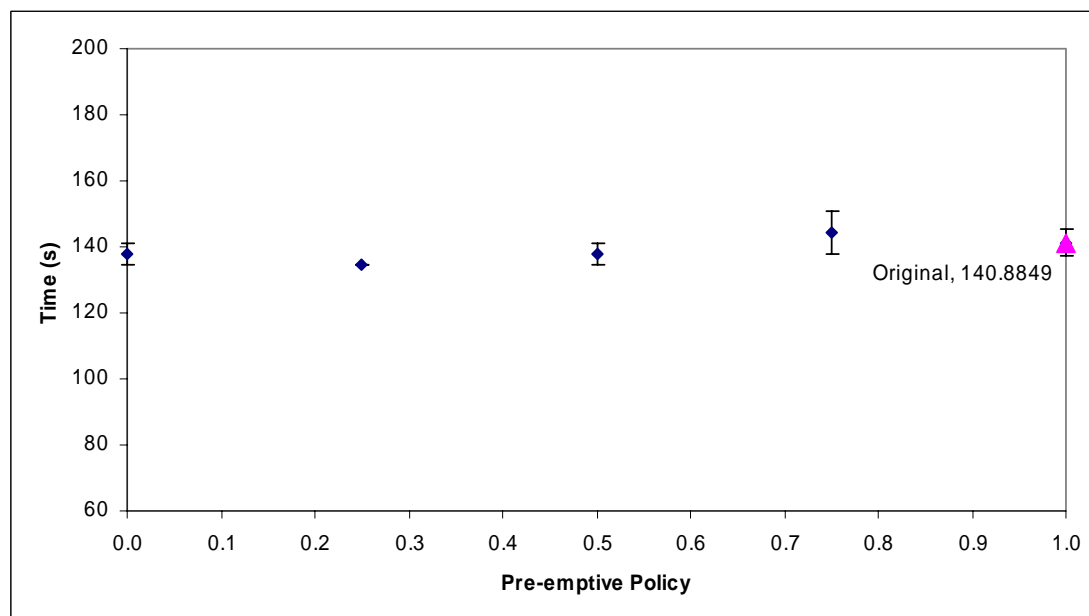


Figure 14: Changing the pre-emptive policy for the chunk cache produced no effect on write times. Results presented are for Phase 2.

## 5.6 Other Buffer Tests

### 5.6.1 Sieve Buffer Size

To fully explore the effect of the sieve buffer, the size of this buffer was changed from 51KB to 8.7MB. The chunk was left at the standard size. The results show no effect using the sieve buffer and as the sieve buffer may not be implemented on all file drivers, it is assumed that this is the case on HPCx.

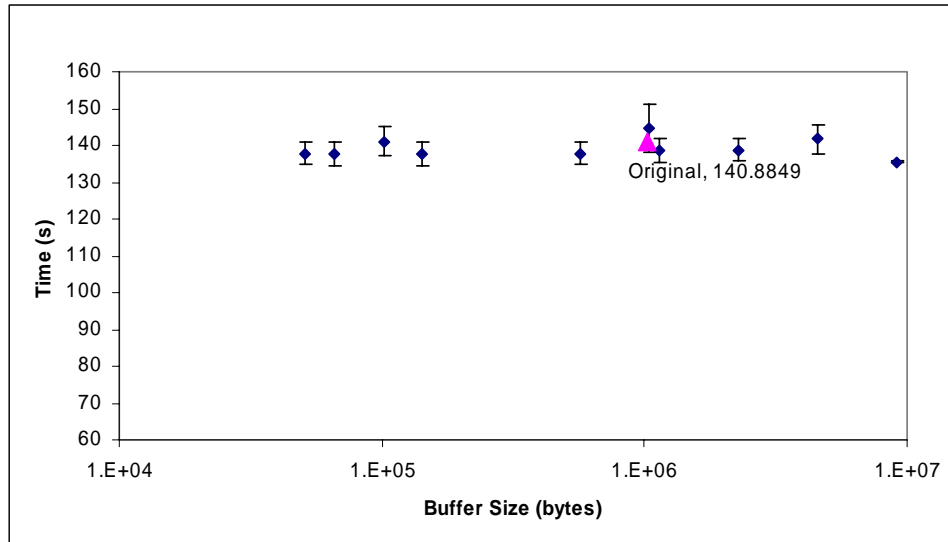


Figure 15: Changing the sieve buffer had no effect on performance. Results presented are for Phase 2.

### 5.6.2 Buffer Size

HDF5 has a background and type conversion buffer, which by default is 1MB in size. The size of this buffer may affect write times on this test code, but as no data type conversions are done its effect may be limited. The buffer was changed in size from 1KB to 18MB. There is no effect of changing this buffer, confirming that it is not used in this case (Figure 16).

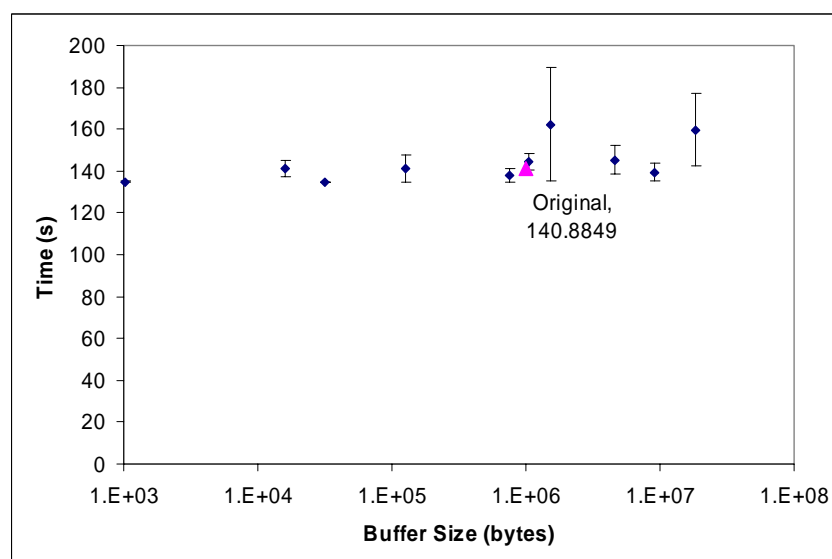


Figure 16: Utilising the data transfer buffer shows no effect on write times. Results presented are for Phase 2.

## 5.7 Compression

### 5.7.1 gzip Compression Level

The level of the `gzip` compression can alter both the file size and write times. Results show the expected outcome with higher compression levels producing smaller files (Figure 17), but longer write times (Figure 18). However, the “step effect” was not expected and cannot be explained at present.

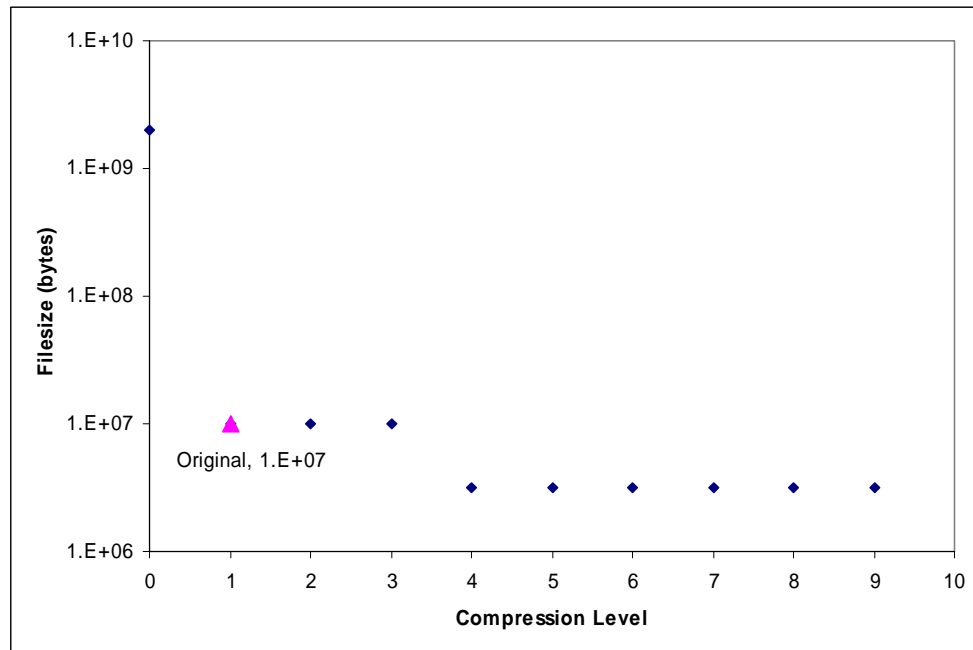


Figure 17: Increasing the compression level decreases the file size as expected, but it does so in a “stepped” fashion. Results presented are for Phase 2.

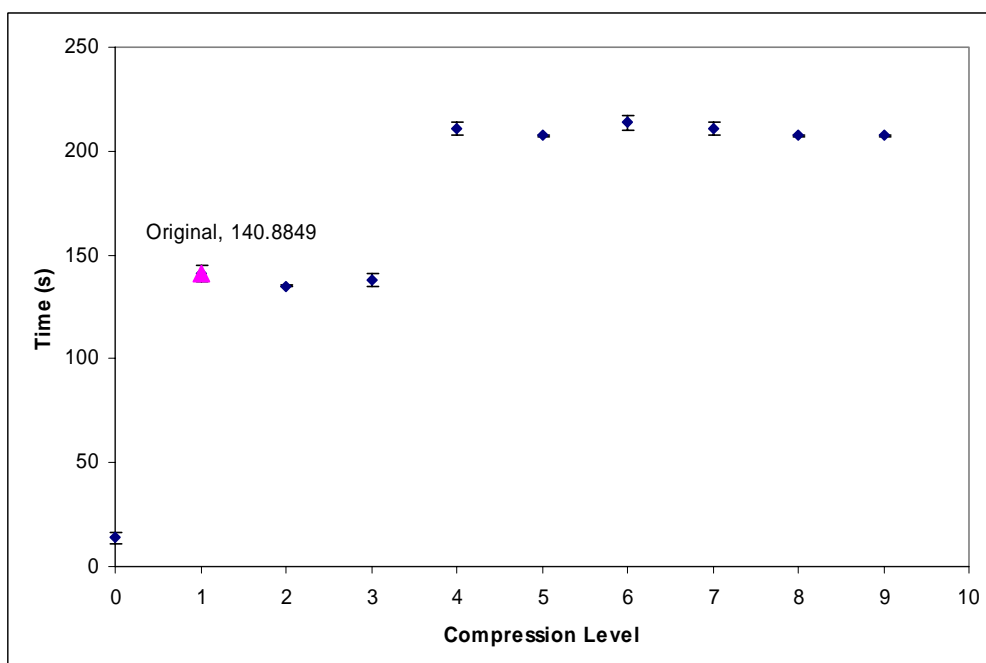


Figure 18: Increasing compression level increases the write times as expected, but as with the file size, it does so in a series of steps. Results presented are for Phase 2.

### 5.7.2 `szip` Compression

An alternative compression algorithm that can be used with HDF5 is `szip`. However, the `szip` functionality is not fully included on HPCx. According to the HDF5 user guide, `szip` can be enabled for decompression only or compression and decompression. In order to try `szip` compression on this data, a local version of HDF5 (version 1.6.4) was compiled with `szip` enabled on Phase 2a. The results were then compared to a standard run from Phase 2, which used `gzip` compression. The results show that using a mask size of 32 not only reduces the time from 85 to 25 seconds (Figure 19) but also decreases the file size from 15MB to 2MB. The optimal pixel size is 8, 10 or 16 bytes. Note that these tests were carried out on Phase 2a, which would account for around 20% of the decrease in write times. However, even taking this into account, using `szip` (with a mask size of 32 bytes) is still significantly faster than the standard run.

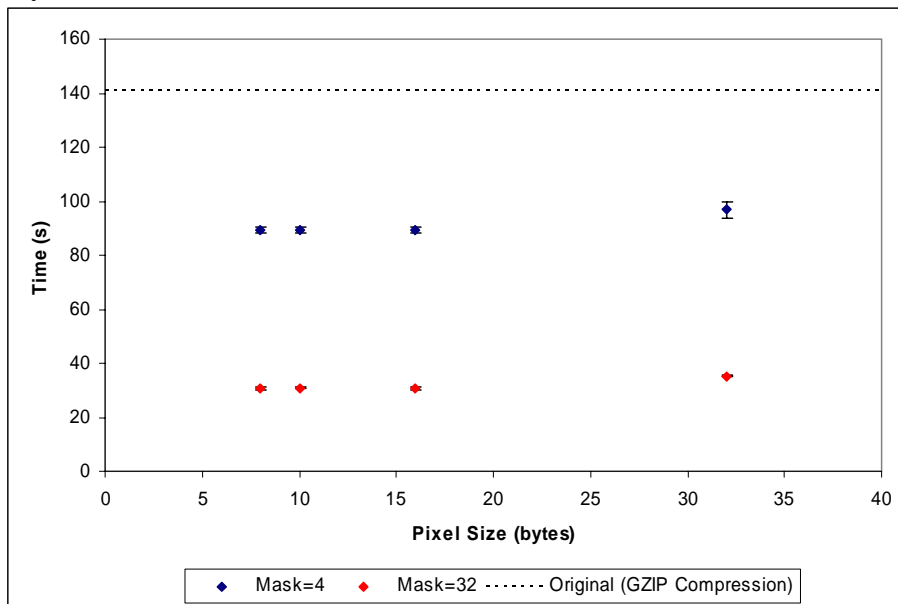


Figure 19: Using `szip` compression improved both the write time and compression ratio. The horizontal dashed line is the time taken for the standard run. Results presented are for Phase 2a.

### 5.7.3 `gzip` Compression

The improved performance observed with `szip` could be due to two factors: either the `szip` algorithm is more efficient than the `gzip` algorithm, or the user installation is more optimal than a system installed compression algorithm. This hypothesis was tested by compiling a user version of the `gzip` library on Phase 2a (

Table 2) and comparing with the system installed version. The user installed version resulted in a write time 76% lower than that of the system installed version. This is equivalent to nearly a five times increase in performance. This suggests that the performance of the system installed `gzip` algorithm on both Phase 2 and Phase 2a is poor and utilising a user installed version offers significant performance improvement. This decrease in write time was the largest over all the tests while using some form of compression, including the increased performance due to the switch to Phase 2a. Only turning off compression completely resulted in a lower write time. In all instances, the `szip`, `gzip` and HDF5 libraries were compiled with the same flags: “-qarch=pwr4 -O3”.

Table 2: Comparison of Phase 2 and Phase 2a using various compression libraries. Times recorded are to the average time to write HDF5 file using the default chunk sizes.

	<b>Phase 2 (s)</b>	<b>Phase 2a (s)</b>	<b>% decrease in write times on Phase 2a</b>
<b>Uncompressed</b>	13.88	13.06	5.91%
<b>gzip (system)</b>	140.88	90.3 <sup>2</sup>	35.91%
<b>szip (user)<sup>3</sup></b>	36.74	25.96	29.34%
<b>gzip (user)</b>	<sup>4</sup>	21.75	75.91% <sup>5</sup>

## 5.8 Other Tests

### 5.8.1 Data Block Size

The data block can be used to store small chunks of data along with other data being written to the file, thereby reducing the number of I/O operations. The size of the data to be stored is used by the HDF5 library to decide if this mechanism of storing data would be advantageous. The default size of data to be stored in this way is 2KB. This was changed from 0 to 131KB to allow HDF5 more opportunity to use this mechanism. Note that the chunk size is a little smaller than the largest data block size used. The results (Figure 20) show no effect in increasing the data block size.

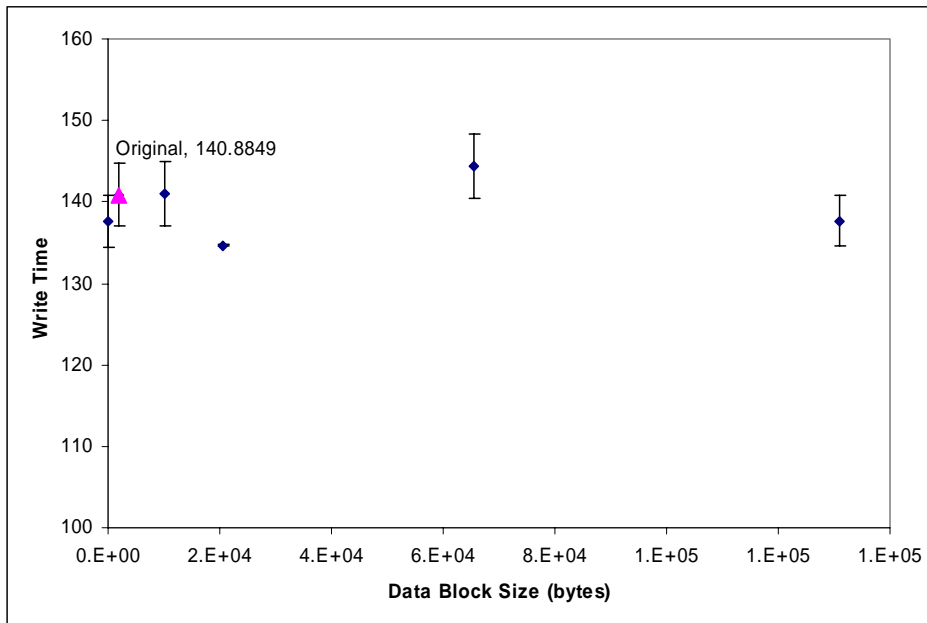


Figure 20: Increasing the data block size produced no discernible effect on write times. Results presented are for Phase 2.

<sup>2</sup> This time is for a 32-bit serial version of HDF5

<sup>3</sup> Same version of szip was used on both Phase 2 and Phase 2a

<sup>4</sup> gzip library not compiled in user space on Phase 2

<sup>5</sup> Comparison is between the system gzip on Phase 2a to the user gzip on Phase 2a

### 5.8.2 Meta Block Size

Meta data is stored in blocks of a given size. The meta data for the OCCAM test data contains items such as the size of data stored, the chunking properties and the endian of the data. Increasing this size of the meta data block can reduce the number of I/O operations, hence increasing performance. The default meta block size is 2KB. This was changed from 0 to 131KB. As with the data block test, there are no discernable effects (Figure 21) on write times when using this parameter.

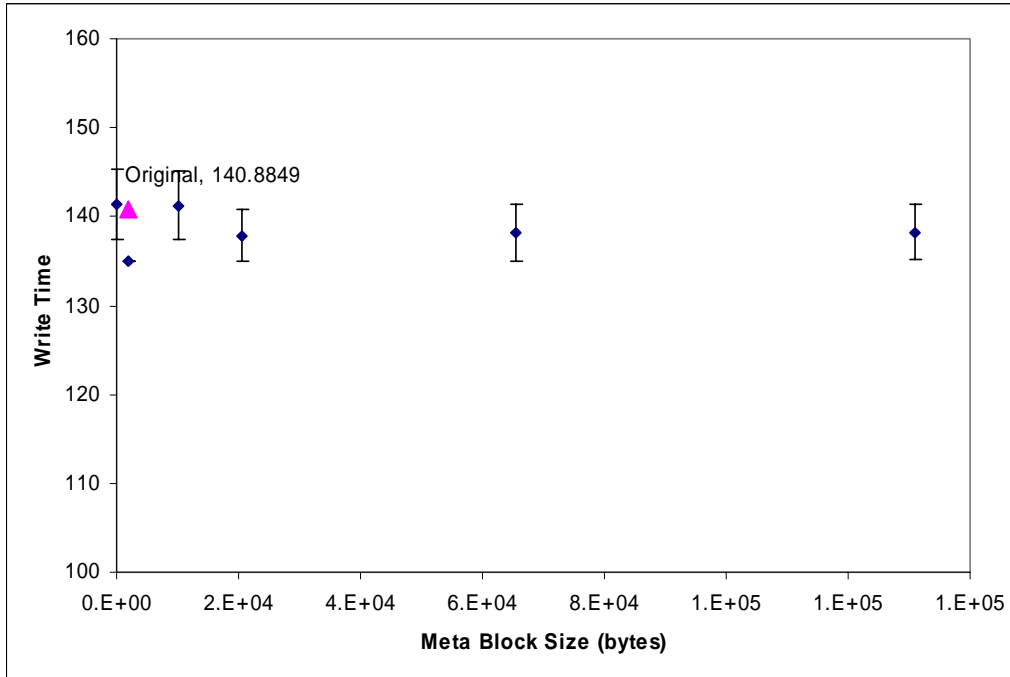


Figure 21: The effect of the meta data block size on the OCCAM data is negligible. Results presented are for Phase 2.

### 5.8.3 Shuffle

Shuffling can affect write times and file size by re-ordering the bytes in the data stream. Implementing a shuffle filter produced an increased write time (14 seconds slower, increasing the time taken to write the file to 154 seconds) and had no effect on file size. However, the actual OCCAM data may behave differently to the generated data used in these tests.

## 5.9 Parallel vs. Serial HDF5

The current HDF5 library on HPCx is an MPI implementation of the library. As such it may occur additional overheads in calling MPI, especially if the file is written in serial. To test if this is the case, a serial version of the library was compiled in the user space. In addition, `szip` encoding and decoding was enabled. As such the comparison between the HDF5 libraries is a 64-bit, MPI-enabled, `szip`-disabled version (installed as a package on HPCx) and a 64-bit, serial, `szip`-enabled version (installed on the user space on HPCx). In addition, due to difficulties with the `gzip` library on Phase 2a, a version of the `gzip` library was also compiled and installed in the user space, rather than use the system version as was the case on Phase2.

Repeating the chunk size test on Phase 2a shows that there is little difference in

performance between a parallel and serial version of HDF5 1.6.4. The serial version was on average 2% quicker over all chunk size tests. Some tests results in an 8% speed-up using the serial library, while one test saw a very small decrease in performance (<0.01%).

### 5.10 64 vs. 32 bit HDF5

As HPCx is a 64-bit computer, it makes sense to compile software in 64-bit. The chunk size test was repeated for 32- and 64-bit serial versions of HDF5 1.6.4 on Phase 2a. The 32-bit library was around 5% slower over all the tests. The maximum decrease in performance was 8%.

## 6 Conclusions

The work carried out on HPCx using HDF5 shows that many of the default parameters of HDF5 give good, robust performance. The main bottleneck for performance was `gzip` compression. A compression level of 1-3 seems optimum, although, it does not matter if it is set to 1, 2, or 3. The performance of `gzip` can be improved by replacing the system version of the `gzip` library with a user compiled one. It was noted during the configuration of HDF5 that the `gzip` library was not being picked by HDF5. A quick solution to this was to compile and install a version of the `gzip` library in the user's directory, alongside the serial version of HDF5. When HDF5 was compiled using this `gzip` library, the performance improved by an average of 575% over all chunk sizes on Phase 2. It would be a reasonable step, if performance of the HDF5 library was not sufficient, for a user to compile their own version of the compression and HDF5 libraries as the source code is freely available. In some cases it will be worth replacing `gzip` with `szip` compression but this is not the case here when using the recompiled `gzip` library. It is important to experiment with the compression levels (in the case of `gzip`) and the pixel and mask size (in the case of `szip`) to find which settings are optimum for any particular dataset.

When using chunked datasets (which one must do with compression enabled), the most important parameter is the size of the chunk cache. This must be at least as big as one chunk. It is advised that the number of cache levels be at least equal to the number of chunks that can fit into the chunk cache. For most purposes, leaving the number of levels at the default level should suffice. Using HDF5 on HPCx benefits from a chunk which is 64KB in size, although the reasons for this are not clear.

All other tests did not decrease write times of the OCCAM dataset, although they may for other forms of data.

To summarise, the optimum parameters for compressed OCCAM data are:

- 64KB chunk size,
- 1MB chunk cache,
- Level 1 compression,
- No shuffling,
- An improved version of HDF5 and `gzip` have been installed on HPCx.

## 7 Acknowledgements

The authors wish to thank Dr. Andrew Coward, National Oceanography Centre, Southampton, for supplying the Fortran program on which this project was based and for his help and assistance.

## 8 References

- [1] Saunders, P. et al, 1999. *Circulation of the Pacific Ocean seen in a global ocean model (OCCAM)*. J. Geophys. Res., 104, C8, 18281-18299.
- [2] The HDF5 Website: <http://hdf.ncsa.uiuc.edu/whatishdf5.html>. Last accessed: 30/09/2005
- [3] NCSA and UIUC, 2001. *HDF5 Wins 2002 R&D 100 Award*. [http://hdf.ncsa.uiuc.edu/HDF5/RD100-2002/All\\_About\\_HDF5.pdf](http://hdf.ncsa.uiuc.edu/HDF5/RD100-2002/All_About_HDF5.pdf). Last accessed: 30/09/2005
- [4] Wei, W, 2005. *Comparison of Portable Binary Data Formats within a Cosmological Simulation*. MSc Dissertation, EPCC, University of Edinburgh.
- [5] NCSA, 2003. *A User's Guide for HDF5, v. 1.4.5*. <http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.user.html>. Last accessed: 30/09/2005
- [6] NCSA, 2003. *Performance evaluation report: gzip, bzip2 compression with and without shuffling algorithm*. [http://hdf.ncsa.uiuc.edu/HDF5/doc\\_resource/H5Shuffle\\_Perf.pdf](http://hdf.ncsa.uiuc.edu/HDF5/doc_resource/H5Shuffle_Perf.pdf). Last accessed: 3/10/2005
- [7] Coward, A, 2003. *OCCAM HDF5 Routines*. <http://www.noc.soton.ac.uk/JRD/OCCAM/LIBOCCAM5>. Last accessed: 30/09/2000
- [8] HDF Group, 2003. *Dataset Chunking Issues*. <http://hdf.ncsa.uiuc.edu/HDF5/doc/Chunking.html>. Last accessed: 07/11/2005

## APPENDIX A Profiling Results

The following profile is the flat profile from `gprof`. The bold rows are those associated with the `gzip` compression.

% Time	Cumulative (s)	Self (s)	Calls	Self (ms/call)	Total (ms/call)	Name
<b>30.7</b>	<b>88.93</b>	<b>88.93</b>				<b>.deflateInit2_ [1]</b>
18.3	141.87	52.94				._log [2]
12.2	177.17	35.30	494568690	0.00	0.00	.grot2_2_1 [6]
11.9	211.75	34.58				._exp [7]
6.8	231.47	19.72				.__mcount [8]
<b>5.8</b>	<b>248.41</b>	<b>16.94</b>				<b>.longest_match [9]</b>
<b>5.6</b>	<b>264.69</b>	<b>16.28</b>				<b>.adler32 [10]</b>
2.2	271.00	6.31	1	6310.00	44553.33	.main [3]
1.5	275.22	4.22				._pxldmod [11]
1.0	278.16	2.94	494683200	0.00	0.00	.grot2 [5]
0.5	279.72	1.56				.H5V_memcpyvv [12]
0.5	281.17	1.45				.qincrement [13]
<b>0.4</b>	<b>282.44</b>	<b>1.27</b>				<b>.deflate_fast [14]</b>
0.4	283.68	1.24				.H5S_hyper_get_seq_list_opt [15]
0.3	284.42	0.74				._tr_align [16]
<b>0.2</b>	<b>285.10</b>	<b>0.68</b>				<b>.compress_block [17]</b>
0.2	285.75	0.65				.qincrement1 [18]
0.2	286.37	0.62				.copy_block [19]
0.2	286.86	0.49				.send_all_trees [20]
0.1	287.21	0.35				.__stack_pointer [21]
0.1	287.40	0.19				.bi_flush [22]
0.1	287.59	0.19				.gen_bitlen [23]
0.1	287.76	0.17				.\$SAVEF31 [24]
0.1	287.91	0.15				.gen_codes [25]

The following profile is the flat profile from `gprof`. No compression was used for this profile.

% Time	Cumulative (s)	Self (s)	Calls	Self (ms/call)	Total (ms/call)	Name
30.1	0.49	0.49				._log [1]
20.9	0.83	0.34				._exp [4]
17.8	1.12	0.29	4276800	0.00	0.00	.grot2 [5]
6.7	1.23	0.11	1	110.0	400.0	chunkwr [2]
6.3	1.33	0.1				.__mcount [6]
5.5	1.45	0.09				._xlcabsv [7]
2.5	1.46	0.04				.H5S_hyper_get_seq_list_opt [8]
2.5	1.5	0.04				.__sqrt_raise_xcp [9]
1.2	1.52	0.02				._sqrt [10]
0.6	1.53	0.01				.H5AC_protect [11]
0.6	1.54	0.01				.H5AC_unprotect [12]
0.6	1.55	0.01				.H5FL_blk_malloc [13]
0.6	1.56	0.01				.H5S_hyper_adjust [14]
0.6	1.57	0.01				.H5TB_ffind [15]
0.6	1.58	0.01				.H5TB_next [16]
0.6	1.59	0.01				.__heap_lock [17]
0.6	1.60	0.01				._pxldmod [18]
0.6	1.61	0.01				.free_y [19]
0.6	1.62	0.01				.global_lock_ppc_mp [20]
0.4	1.63	0.01				.qincrement1 [21]

## APPENDIX B Phase 2a Results

Due to difficulties with the `gzip` library on Phase 2a, a version had to be compiled on the user space. This version of the library proved to be much quicker than that on Phase 2, which makes comparisons between the two systems impossible when using `gzip` compression. However, `szip` using a 32-bit serial version of HDF5 was used on both Phase 2 and Phase 2a. The results are shown in the following two figures. In both instances, both the `szip` and HDF5 libraries were compiled with the same flags: “`-qarch=pwr4 -O3`”.

