

# NetCDF-4 Performance Report

Choonghwan Lee ([cholee@hdfgroup.org](mailto:cholee@hdfgroup.org))

MuQun Yang ([ymuqun@hdfgroup.org](mailto:ymuqun@hdfgroup.org))

Ruth Aydt ([aydt@hdfgroup.org](mailto:aydt@hdfgroup.org))

The HDF Group (THG)

June 9, 2008

## 1. Introduction

NetCDF-4 [1] is an I/O software package that retains the original netCDF [2] APIs while using HDF5 [3] to store the data. Sponsored by NASA's Earth Science Technology Office [4], netCDF-4 is the result of a collaboration between Unidata [5] and The HDF Group [6].

The netCDF-4 project has generated enormous interest in both the netCDF and HDF user communities, and substantial reductions in data storage space have been achieved with the netCDF-4 beta releases using the HDF5 compression options. The performance of netCDF-4 is also a critical factor for users of netCDF-3 (the latest version of netCDF) who are considering a move to netCDF-4. In particular, these users may ask:

- How does the performance of netCDF-4 compare to that of netCDF-3?
- Under what circumstances can an application get better performance with netCDF-4? How can performance be optimized?
- Under what circumstances can an application get poor performance with netCDF-4? What can be done to avoid poor performance?

Through the use of benchmark and examples, this report addresses these questions and helps users gain an understanding of the performance characteristics of netCDF-4. Sections 2 through 5 contain detailed information and Section 6 summarizes the findings. The report includes:

- §2 Description of Benchmark and Test Environment
- §3 NetCDF-3 and NetCDF-4 Performance Comparisons
- §4 Performance Tuning with NetCDF-4
- §5 Performance Pitfalls with NetCDF-4
- §6 Conclusions

Although netCDF-4 supports parallel I/O via MPI-IO, this report only discusses sequential I/O.

## 2. Description of Benchmark and Test Environment

The netCDF-3 and netCDF-4 performance comparison results are based on an I/O benchmark originally developed at Unidata. The original benchmark included data input (data read) performance comparisons between netCDF-3 and netCDF-4. The Unidata benchmark was rewritten by The HDF Group to include data output (data write) operations and to allow for more control over various parameters within the benchmark.

## 2.1. Timing Methodology

The benchmark uses the C function *gettimeofday* to bracket netCDF *nc\_get\_vara* and *nc\_put\_vara* read and write functions, and computes the elapsed wall clock time to read or write a variable. The reported data read rate is the size of the array variable (in megabytes) divided by the elapsed wall clock time for the associated *nc\_get\_vara* call. The write data rate is calculated similarly. It is worth noting that while the netCDF terms “variable” or “array variable” are used almost exclusively in this report, the equivalent HDF5 term is “dataset”.

By default, the Linux kernel caches programs and data in memory as long as possible. Therefore, unless special measures are taken, the elapsed wall clock I/O time may reflect the time to access memory rather than disk (or, more accurately, the I/O subsystem). Benchmarks run under these conditions may be good predictors for applications with problem sizes whose input and output can be cached in the system memory. Several of the benchmark runs were made under these default system caching conditions.

The remaining benchmark runs were structured to provide more accurate I/O performance information for applications with problem sizes that will not fit in the system cache, that read data only once and therefore do not benefit from caching, or that have small problem sizes but run on heavily-used shared memory systems. For these runs, every netCDF *nc\_put\_vara* call was followed by *ncsync* and *fsync* calls to ensure the data was written to disk. The closing *gettimeofday* call followed the *fsync*, so the computed elapsed time included the data transfer to disk. Also in this set of runs, the Linux tunable */proc/sys/vm/drop\_caches* [7] was used to instruct the kernel to clear non-dirty objects from the cache prior to the start of a run. This step guaranteed that data would be read from disk and not from system memory.

## 2.2. Array Variables and Execution

The array variables used in the netCDF-3 and netCDF-4 performance comparison experiments reported in Section 3 are shown in Table 1. 32-bit integer array variables ranging from one to six dimensions were used. In the “small” tests, each variable is approximately 1 megabyte in size, while in the “large” tests the variables range from approximately 30 to 40 megabytes. Each variable was written to a separate file.

Variable Name	Number of Dimensions	Dimension Layout Small Tests	Dimension Layout Large Tests
Variable 1	1	[262144]	[10000000]
Variable 2	2	[512][512]	[3162][3162]
Variable 3	3	[64][64][64]	[215][215][215]
Variable 4	4	[22][22][22][22]	[56][56][56][56]
Variable 5	5	[12][12][12][12][12]	[25][25][25][25][25]
Variable 6	6	[8][8][8][8][8][8]	[14][14][14][14][14][14]

**Table 1: Array variables for performance comparisons.**

For the netCDF-3 and netCDF-4 performance comparisons reported in Section 3, an entire array variable was written or read at once. Each benchmark run measured the time to read or write a single array variable.

The benchmark execution sequence was:

Run 1: Open File1; write Variable 1; close File1  
Run 2: Open File1; read Variable 1; close File1  
Run 3: Open File2; write Variable 2; close File2  
Run 4: Open File2; read Variable 2; close File2  
...  
Run 11: Open File6; write Variable 6; close File6  
Run 12: Open File6; read Variable 6; close File6

The entire benchmark sequence, consisting of 12 runs alternating between writes and reads, was executed ten times for each unique experimental configuration. *The best and worst cases were dropped, and the reported results are the average of the eight remaining executions.*

For configurations with the default system caching, it is expected that variables being read (in runs 2, 4, ...) would be in the system cache, as they would have just been written by the previous test (in runs 1, 3, ...). Caching or pre-fetching done by the netCDF-3 or netCDF-4/HDF5 library read operations should not affect the read performance in these tests, as each run issues a single read request.

The I/O libraries have some variation in the underlying operations performed when a file is opened, written, read, and closed. The benchmarks presented focus on the write and read operations. Instrumentation of all calls to the I/O library would give a more complete picture of the total time devoted to I/O-related activities.

### **2.3. Test Environment**

All benchmarks were run using netCDF-4 version netCDF 4.0-snapshot2008012502. This version is based on HDF5 1.8 subversion revision 14468, which is essentially the same as HDF5 release 1.8.0. The first production netCDF-4 release, scheduled for June 2008, will be based on HDF5 release 1.8.1. No substantial performance differences are expected between the versions tested and the production release. NetCDF 3.6.2 was used to examine the netCDF-3 performance.

The benchmarks were run on two Linux machines. The first machine, referred to as the “Intel-based machine”, had two Intel Xeon 3.2GHz CPUs, 8 GB of memory, the gcc 3.4 compiler, and was running Linux 2.6.9-42.0.10. The disk configuration was a RAID level 0 array of 5 Maxtor 6B300S0 SATA disks (7200 RPM, 1.5 Gb/s, 16 MB buffer, 4.1 ms latency, 9.3 ms seek). The *drop\_caches* tunable was introduced in Linux 2.6.16, and therefore was not available on the Intel-based machine.

The second test machine, referred to as the “AMD-based machine”, had two dual-core AMD Opteron 2.4GHz CPUs, 8 GB of memory, the gcc 4.1.2 compiler, and was running Linux 2.6.16.54-0.2.5. The disk used in the tests was a WD Raptor SATA drive (10,000 RPM, 1.5 Gb/s, 16 MB buffer, 2.99 ms latency, 5.2 ms seek). The *drop\_caches* tunable was available on the AMD-based machine.

Because of time and resource constraints, the benchmark executions were split across the two machines just described. All runs with the system default caching behavior were done on the Intel-based platform. All runs that used *drop\_caches* and *ncsync/fsync* to override the default system caching behavior and force I/O to disk were made on the AMD-based system.

In reviewing the benchmark results, it is important not to directly compare results from runs made on different platforms, but rather to compare results on the same platform with different configuration options. The primary purpose of this report is to provide insights into the behavior of netCDF-4 with various user-settable configurations, and not to carry out cross-platform comparisons.

### 3. NetCDF-3 and NetCDF-4 Performance Comparisons

Applications built with netCDF-3 can only store or retrieve data contiguously. With netCDF-4 the application may access data contiguously and also has the option of dividing the data into equally-sized *chunks* that are stored and retrieved separately. Using the array variables and execution sequence described in Section 2.2, the performance of netCDF-3 using contiguous storage is compared to the performance of netCDF-4 using both contiguous and chunked storage.

#### 3.1. Contiguous Storage

In this section the performance of netCDF-4 using contiguous storage is compared to the performance of netCDF-3, which always uses contiguous storage.

##### 3.1.1. Default System Caching

The first set of experiments was run on the Intel-based machine with the system default caching behavior. The write and read performance results for the six array variables are shown in Figure 1. Results for both the small and the large tests appear in the figure.

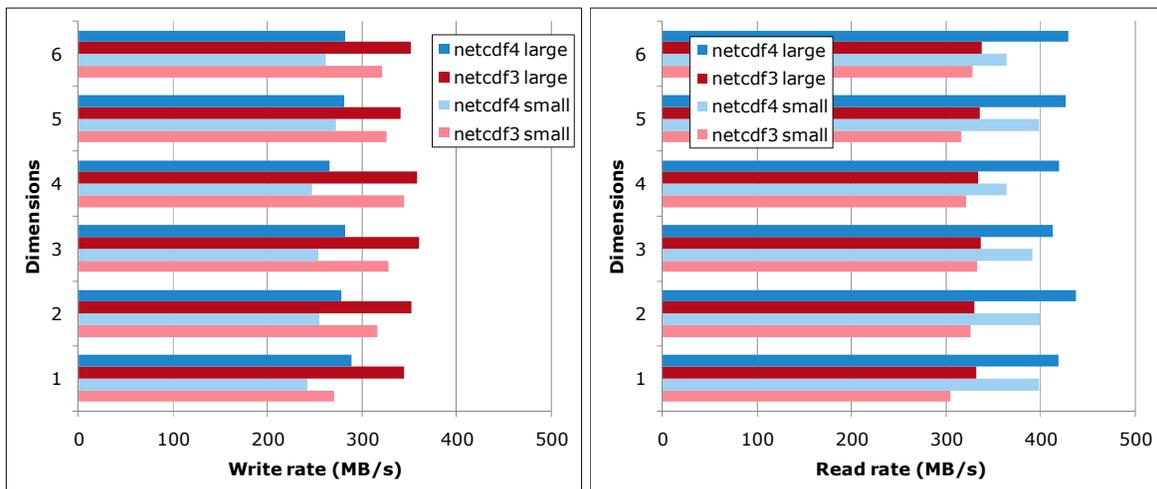


Figure 1: NetCDF-4 contiguous storage, default system caching, Intel-based machine.

While netCDF-4 introduces an additional layer of function calls to the HDF5 library, it outperformed netCDF-3 for the read experiments. On the other hand, netCDF-4 was slower for writes. The poorer write performance could be due to the data buffering used by the underlying HDF-5 library with the contiguous storage layout, but further tests are needed to fully understand the cause.

### 3.1.2. Override of Default System Caching

The second set of experiments was run on the AMD-based machine. *ncsync* and *fsync* calls were made after each write to flush the data to disk, and *drop\_caches* was called prior to each read run to clear the system cache.

As shown in Figure 2, the write performance of netCDF-3 and netCDF-4 was very similar overall. There was less consistency in the read performance of both netCDF-3 and netCDF-4 in this set of experiments. That said, netCDF-3 significantly outperformed netCDF-4 on reads of large array variables in all dimensions when comparing the average MB/s of the “middle eight” out of ten tests—the results plotted in Figure 2.

Looking at the large read benchmark results for all ten runs per dimension, the maximum transfer rate for netCDF-4 was better than the maximum transfer rate for netCDF-3 for three of the six dimensions. However, netCDF-4 had more slow runs overall, which pulled down the average rate. For this benchmark, netCDF-3 delivered higher read rates on the large tests more consistently than netCDF-4. Further investigation is needed to fully understand the variability of the netCDF-4 results.

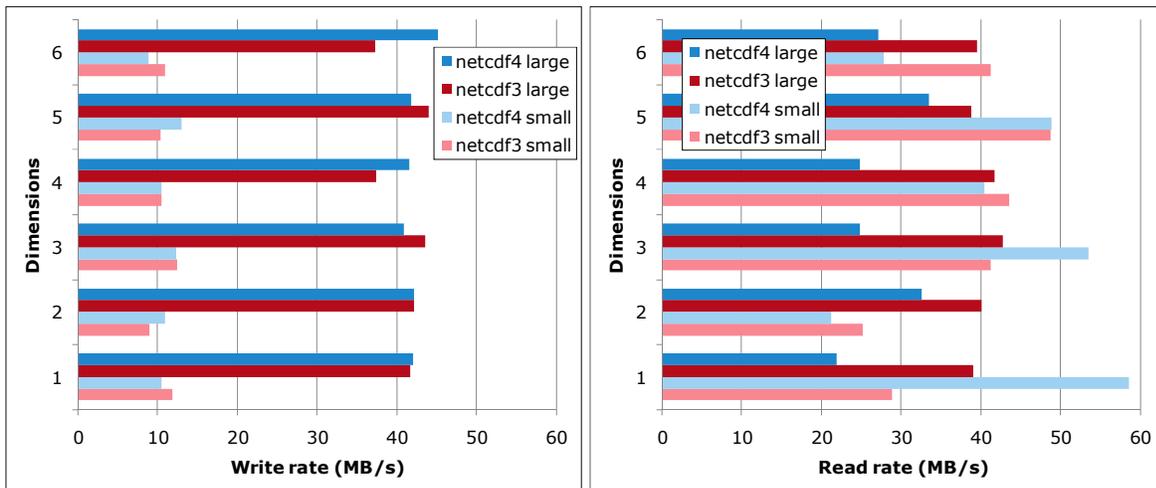


Figure 2: NetCDF-4 contiguous storage, system cache cleared and *ncsync*/*fsync* used, AMD-based machine.

### 3.1.3. Contiguous Storage Summary

With the array variables and execution sequence described in section 2.2 and the default system caching behavior, netCDF-3 was consistently faster than netCDF-4 for writes and consistently slower for reads. When *ncsync*/*fsync* and *drop\_caches* were used to force the I/O subsystem to be accessed instead of system memory buffers, the performance between netCDF-3 and netCDF-4 was comparable for writes; netCDF-3 performed significantly better for large read tests and the performance was inconsistent for the small read tests.

## 3.2. NetCDF-4 Chunked Storage

In this section the performance of netCDF-4 using chunked storage is compared to the performance of netCDF-3 using contiguous storage. An overview of chunked storage is presented prior to the benchmark results.

### 3.2.1. Overview of Chunked Storage Layout

The chunked storage layout is a feature of the HDF5 library that is available to applications using netCDF-4. While chunked storage is typically considered an advanced feature, it is one that can have a significant positive—or negative—impact on data access rates. For that reason, chunked storage receives extensive attention in this performance report.

With a chunked storage layout, a dataset (an array variable in the benchmarks) is partitioned by the HDF5 library into fixed-size pieces that are transferred independently of each other to and from the disk. The fixed-size pieces are referred to as *chunks*. Chunks are related to the physical storage of the data on the disk, not to the logical relationship of data points within the array.

The application can set the chunk size when the dataset (array variable) is created; there are no restrictions on chunk size. The individual chunks can be placed anywhere on the physical storage media since each is transferred separately. The HDF5 library handles all chunk management, making the application’s access to the elements of the array variable identical regardless of whether a chunked or contiguous storage layout is used.

In some cases, chunked storage can provide faster access to subsets of the data, as will be seen in Section 4.1. Chunked storage is required when HDF5’s data filters, such as the compression filter discussed in Section 4.2, are used. Chunked storage is also required when using array variables with extensible dimensions.

Chunked storage may, or may not, offer a performance benefit. A number of factors including the chunk size, the application’s data access pattern, and HDF5’s caching with chunked storage all influence the performance. The performance of chunked storage for the benchmarks described in Section 2.2 is examined next.

### 3.2.2. Default System Caching

In these experiments, run on the Intel-based machine with default system caching, the netCDF-4 chunk size was set to the size of the array variable. For example, a chunk size of 512×512 was used for the 2D Variable 2 in the small test case. Contiguous storage was used for netCDF-3.

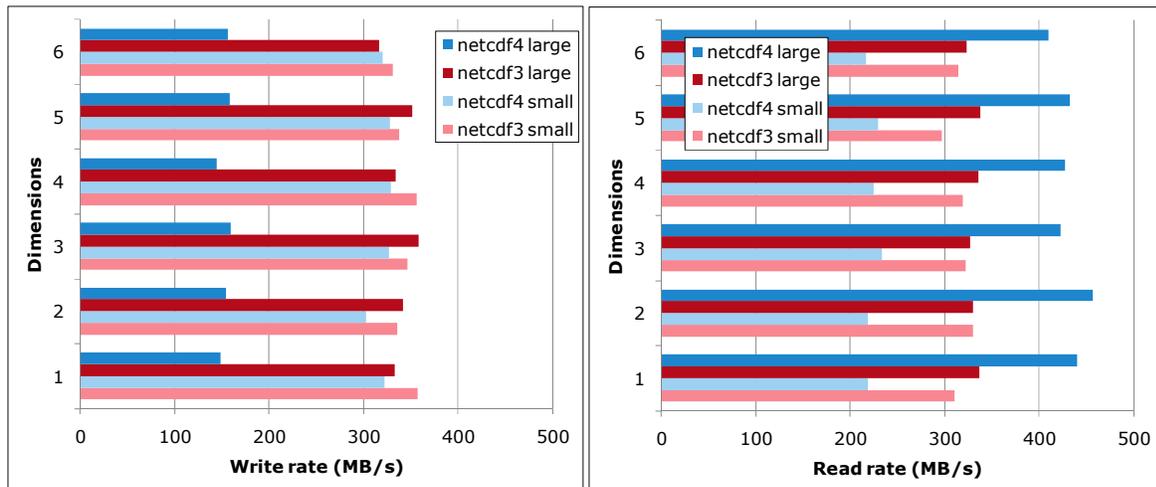


Figure 3: NetCDF-4 chunked storage with one chunk per variable, default system caching, Intel-based machine.

As in the previous experiments, the benchmark application writes or reads an entire array variable with a single API call. The write and read performance results for the six array variables are shown in Figure 3.

The results in this set of experiments were fairly consistent across all array dimensions. NetCDF-3 was more than twice as fast as netCDF-4 for writes in the large tests and substantially better for reads in the small tests. NetCDF-4 was substantially better than netCDF-3 for reads in the large tests. The performance of netCDF-3 and netCDF-4 was similar for writes in the small tests.

The poor performance of netCDF-4 when writing large array variables in these experiments led The HDF Group to conduct further investigations. Two factors in the HDF5 library were identified as contributing to the slow write rates measured for the large tests, and further experiments were conducted to assess the impact of those factors.

The first factor was the storage space allocation policy. The default policy with chunked storage is to allocate space incrementally on a chunk-by-chunk basis as data is written to the array cells in a given chunk. The *H5Pset\_alloc\_time* routine was used to override the default policy with one that allocates space when the dataset is created. This change increased the write rate by over 100 MB/s.

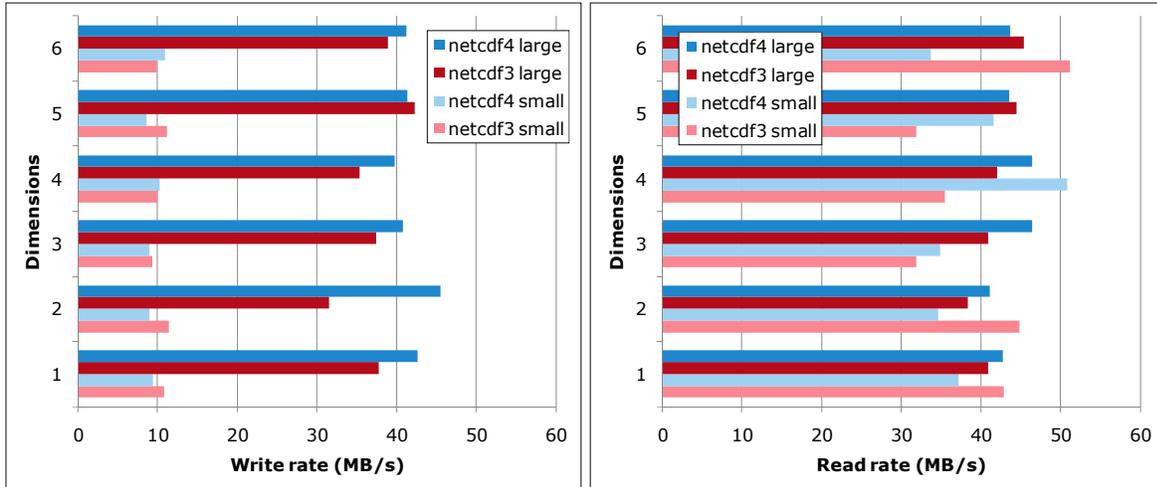
The second factor involves the HDF5 cache, whose default size in HDF5 is 1 MB. This size is sufficient for the small array variables, but not for the large test cases. Using the HDF5 *H5Pset\_cache* routine to increase the HDF cache size to 64 MB improved the measured write rate by over 200 MB/s. It should be noted that while the measured write rate increased with the larger cache size, the overall runtime may not decrease by an equal amount, as the HDF5 cache will be flushed when the file is closed. Comparison tests of overall runtime were not conducted.

While users of netCDF-4 are currently not able to take advantage of the HDF5 tuning routines *H5Pset\_alloc\_time* and *H5Pset\_cache*, this example shows there is room for improvement in the performance of the initial netCDF-4 release. The HDF Group will examine performance results such as these to identify and remedy HDF5 library bottlenecks in future releases, and will continue to work with Unidata to optimize how netCDF-4 uses the HDF5 library.

Returning to the netCDF-4 chunked storage results shown in Figure 3, and comparing them to the netCDF-4 contiguous storage results shown in Figure 1, contiguous storage outperformed chunked storage for small reads, and was comparable for large reads. This is likely due to the additional management overhead incurred with chunked storage, which is especially noticeable with smaller sized variables. Write rates for the small tests with chunked storage were better than with contiguous storage, likely due to considerable optimization in the HDF5 code for chunked writes. However, write rates for the large tests were much worse with chunked storage, probably due to the storage allocation policy and HDF5 cache size discussed earlier.

### **3.2.3. Override of Default System Caching**

In the fourth set of experiments, the netCDF-4 chunk size was again set to the size of the array variable, and netCDF-3 again used contiguous access. Experiments were run on the AMD-based machine. *ncsync* and *fsync* calls were made after each write to flush the data to disk, and *drop\_caches* was called prior to each read run to clear the system cache. The measured I/O performance is shown in Figure 4.



**Figure 4: NetCDF-4 chunked storage with one chunk per variable, system cache cleared and ncsync/fsync used, AMD-based machine.**

As was the case with forced disk access and contiguous storage in experiment set 2 (see Section 3.1.2), the I/O rates vary considerably for both netCDF-3 and netCDF-4 across the dimensions, especially for reads. Overall, netCDF-3 and netCDF-4 performed comparably for both reads and writes. NetCDF-4 was better for five of the six large write tests, while netCDF-3 was better for four of the six small write tests. In contrast to the very poor performance of netCDF-4 when writing the large array variables with cached storage and default system caching (see Section 3.2.2), the results here with forced disk access are more in line with what netCDF-3 delivers.

Comparing the netCDF-4 transfer rates in Figures 2 and 4, chunked storage consistently performed better than contiguous for the large read tests. In all other cases, performance was similar or—in the case of small reads—difficult to compare due to variability across dimensions.

### 3.2.4. Chunked Storage Summary

With the array variables and execution sequence described in Section 2.2 and the default system caching behavior, netCDF-3 was consistently faster than netCDF-4 for large writes and for small reads. NetCDF-4 was faster for large reads. When *ncsync/fsync* and *drop\_caches* were used to force the I/O subsystem to be accessed instead of system memory buffers, the performance of netCDF-3 and netCDF-4 was comparable overall. NetCDF-3 was slightly better for the small array variables, while netCDF-4 was slightly better for the large array variables.

## 3.3. Performance Comparisons Summary

The results of the netCDF-3 and netCDF-4 performance tests are summarized in Table 2. The write and read rates are the average rates across all six array variable dimensions that were shown separately in the bar graphs of Figures 1-4. As with all performance results in this section, the best and worst cases for each unique experimental configuration (storage layout, system caching configuration, test array size, dimension of array variable, netCDF version, read or write) were dropped, and the reported results are the average of the remaining executions. Cells are highlighted for configurations where either netCDF-3 or netCDF-4 outperformed the other version by more than 5%.

Storage Layout	System Caching	Figure	Test Array	Write				Read			
				Rate (MB/s)		Best	% Faster	(Rate MB/s)		Best	% Faster
				netCDF3	netCDF4			netCDF3	netCDF4		
Contiguous	Default	1	Large	351	279	netCDF3	26%	334	424	netCDF4	27%
			Small	317	255	netCDF3	25%	321	385	netCDF4	20%
	Override	2	Large	41	42	netCDF4	3%	40	27	netCDF3	47%
			Small	11	11	netCDF4	4%	38	42	netCDF4	9%
Chunked	Default	3	Large	339	153	netCDF3	121%	331	431	netCDF4	30%
			Small	343	321	netCDF3	7%	315	223	netCDF3	41%
	Override	4	Large	37	42	netCDF4	13%	42	44	netCDF4	5%
			Small	10	10	netCDF3	9%	40	39	netCDF3	2%

**Table 2: Summary of netCDF-3 and netCDF-4 performance comparison tests. Reported rates are the average over all six variable dimensions. “Best performer” highlighted if more than 5% difference between netCDF-3 and netCDF-4.**

It should be noted that even “identical” configurations showed some variation across sets of tests. In particular, there is no chunked storage for netCDF-3 so the rates for contiguous and chunked tests of netCDF-3 should be very close. Looking at large writes when system caching is overridden, a 10% difference is noted (41 MB/s versus 37 MB/s). This is not unexpected for tests run on quiet, but not idle, systems where disk access is involved. Since these tests are meant to inform users, not to test optimal performance, the test conditions are reasonable.

Additional in-depth studies are needed to fully explain the benchmark results. The exceptionally poor performance of netCDF-4 for large tests with the default system caching and chunked storage is attributable, in part, to the default storage allocation policy discussed in Section 3.2.2, which should be addressed in a future release.

One factor that contributes to I/O performance is data type conversion. NetCDF-4 can read and write data using the machine’s native data type. On little-endian machines, such as the Intel and AMD systems used in these tests, netCDF-4 will not convert the data to big-endian representation for storage and revert to little-endian for loading, as is done by netCDF-3. The impact of the conversion will vary depending on the size of the data, the speed of the processor, and the overall data transfer time. The time spent on data type conversion was not directly measured in these benchmarks.

For the benchmarks covered in Section 3, netCDF-4 contiguous storage outperformed, or was comparable to, netCDF-4 chunked storage except for: (1) small writes with the default system cache and (2) large reads when the default system caching was overridden and access was forced to disk. For applications that do not need the special features enabled by chunked storage, contiguous storage is probably a better choice.

The netCDF-3 and netCDF-4 performance for contiguous storage was comparable overall, with each claiming the “best performer” title for several of the configurations tested (dimension of array variable, test array size, system caching, read or write). NetCDF-3 outperformed netCDF-4 for writes with default system caching, but netCDF-4 performed better for reads. NetCDF-3 performed better on the large read tests when access was forced to disk, but netCDF-4 did better for the small read tests under those conditions.

On the whole, users considering a move from netCDF-3 to netCDF-4 should see comparable performance for applications using contiguous storage with a balanced number of reads and writes.

Performance-sensitive applications are encouraged to conduct a performance test with their own data, as the benchmarks show variations even for similar-sized array variables with different dimensions. Timing complete runs, rather than concentrating on I/O rates as was done here, is a reasonable approach that will also encompass performance differences between netCDF-3 and netCDF-4 beyond the read and write APIs.

## 4. Performance Tuning with NetCDF-4

NetCDF-4, through the use of HDF5 features, can tremendously improve the I/O performance for some applications. Two cases where HDF5 features can help netCDF-4 users are covered in this section.

### 4.1. Non-contiguous Access Patterns

The first case where HDF5 features can help improve I/O performance involves applications whose access patterns are orthogonal to the normal (contiguous) storage order of the data. An examination of such access patterns is presented in this section.

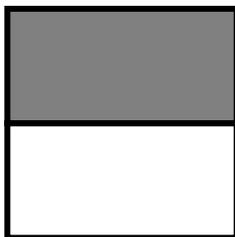
#### 4.1.1. Hyperslabs and Access Patterns

The term *hyperslab* refers to a subset of the data points in an array variable that are accessed simultaneously by an application. All hyperslabs discussed in this report are made up of logically adjacent collections of points. Hyperslabs are related to the logical relationship of data points within the array variable, not to the physical storage of the data on the disk. While 2D array variables and hyperslabs are used in the examples, the principles are relevant for any number of dimensions, hence the name *hyperslab* rather than simply *slab*.

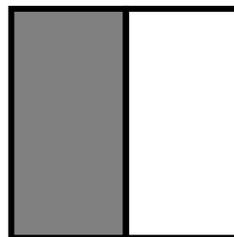
Figure 5a depicts the case where an entire  $512 \times 512$  2D array variable is covered by a single hyperslab, and corresponds to the experiments in Section 3.2. Figures 5b and 5c depict cases where only half of the elements in the 2D array variable are accessed simultaneously. The hyperslabs contain logically adjacent points in all three cases.



**Figure 5a:** 512x512 hyperslab covering all of 2D array variable; contiguous access in row-major order.



**Figure 5b:** 256x512 hyperslab covering half of 2D array variable; contiguous access in row-major order.



**Figure 5c:** 512x256 hyperslab covering half of 2D array variable; non-contiguous access in row-major order.

Even when all points in the hyperslabs are logically adjacent, they may not be physically adjacent on disk. If contiguous storage is used and array elements are saved in row-major order, as they are with netCDF-3 and netCDF-4, then array element (1,1) is physically adjacent to element (1,2), which is physically adjacent to (1,3), and so on. Looking at the end of one row in the array variable and the beginning of the next row, element (1,512) is physically adjacent to element (2,1), element (2,512) is physically adjacent to element (3,1), and so on.

Figures 5a and 5b depict hyperslabs with contiguous access patterns—the data elements making up the subsets of the array variable being accessed by the application are physically adjacent to each other on the disk when contiguous storage is used. Figure 5c, however, depicts a hyperslab with a non-contiguous access pattern—data element (1,256) is not physically adjacent to element (2,1), element (2,256) is not physically adjacent to element (3,1), and so on.

The use of hyperslabs with non-contiguous access patterns can result in poor I/O performance with a contiguous storage layout.

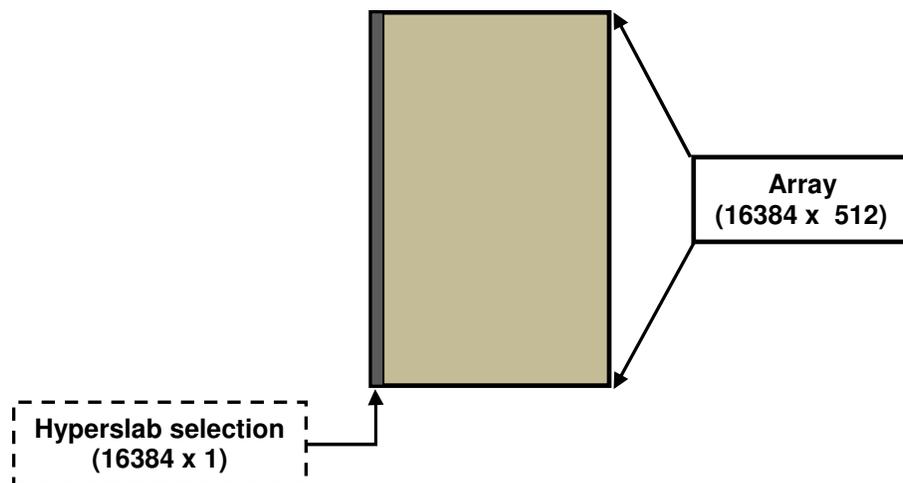
#### 4.1.2. Non-contiguous Access Patterns and Chunked Storage

The following example shows how HDF5’s chunked storage can be used to improve I/O performance for hyperslabs with non-contiguous access patterns.

Consider an integer array with two dimensions. The number of elements in the first dimension is 16384 and the number of elements in the second dimension is 512. For access purposes, consider a hyperslab with 16384 elements in the first dimension and 1 element in the second dimension. This example is summarized in Table 3, and the logical layout of the described array and hyperslab are illustrated in Figure 6 (not to scale). The hyperslab has an extremely non-contiguous access pattern.

Parameter	Value
Array dimensions	2
Array dimension sizes	[16384][512]
Hyperslab dimension sizes	[16384][1]

**Table 3: Example with non-contiguous access pattern.**



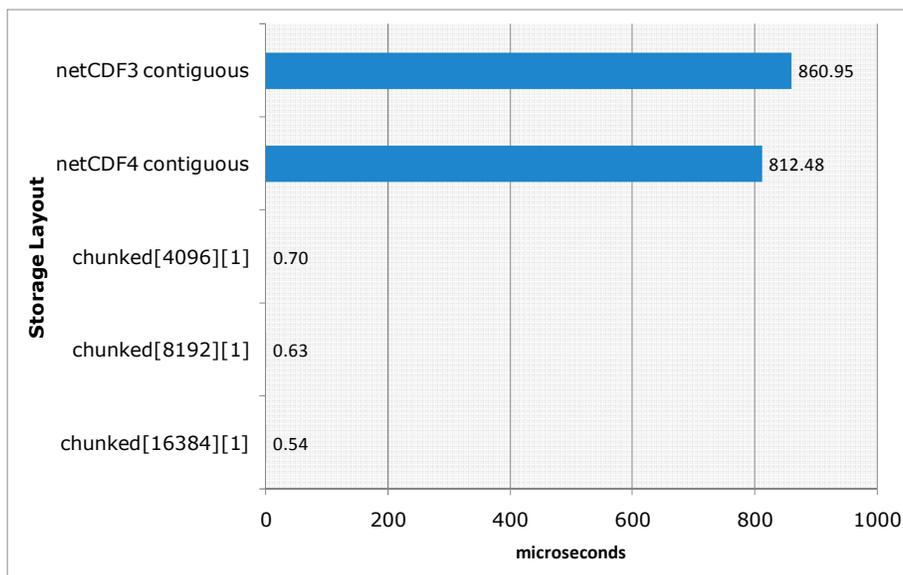
**Figure 6: Logical layout of array and hyperslab selection.**

A variety of contiguous and chunked storage layout options can be used. Recalling the presentation of chunked storage in Section 3.1.1, chunked storage partitions the array into fixed-size pieces that are transferred independently of each other to and from the disk by the HDF5 library. Chunked storage can be used to arrange the array elements on disk in a manner that is better suited for applications with non-contiguous access patterns.

Using the methodology described in Section 2, runs of the non-contiguous access pattern example shown in Table 3 were made on the AMD-based machine with the system cache cleared. Ten runs were made for each of the storage layout configurations presented in Table 4, and the best and worst runs for each layout configuration were dropped. Figure 7 reports the average time to read a single non-contiguous hyperslab (16384×1) based on the eight remaining executions for each storage layout.

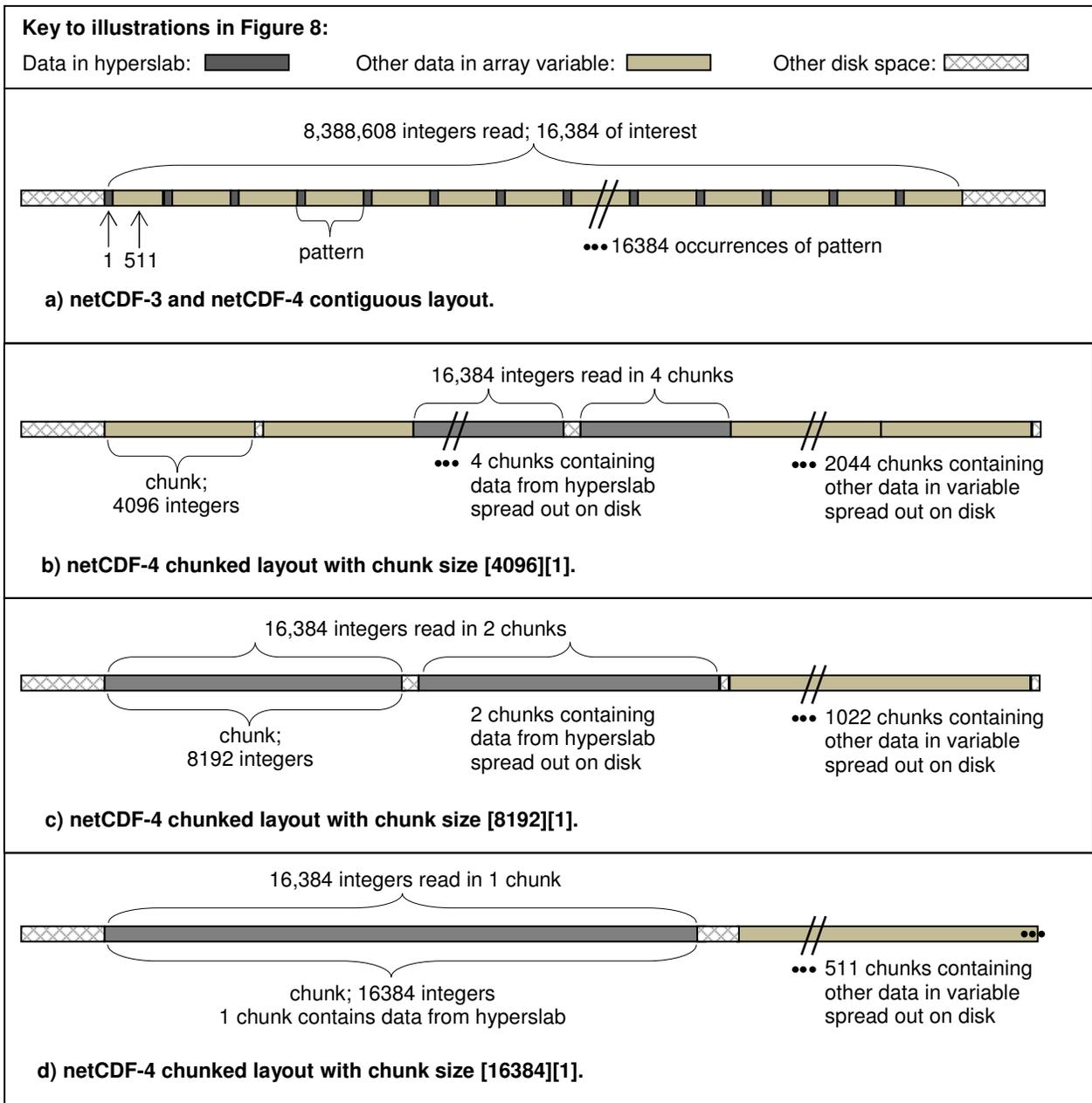
I/O Package	Layout	Chunk size
netCDF3	Contiguous	N/A
netCDF4	Contiguous	N/A
netCDF4	Chunked	[4096][1]
netCDF4	Chunked	[8192][1]
netCDF4	Chunked	[16384][1]

**Table 4: Storage layouts for the 16384 x 512 array.**



**Figure 7: Time to read non-contiguous (16384×1) hyperslab on AMD-based system with system cache cleared.**

The results in Figure 7 show that the netCDF-4 chunked storage layout can deliver significantly better performance than the contiguous storage layout when the access pattern is non-contiguous. In the example given, when the chunk size is 16384×1, an exact match for the 16384×1 hyperslab, the read performance of the netCDF-4 chunked storage layout was more than 1,500 times better than with netCDF-3. The netCDF-4 chunked storage layout was over 24 times faster than netCDF-3 when writing the hyperslab (write results not shown in detail).



**Figure 8: Depiction of various storage layouts on physical disk. For contiguous layout, entire array variable is transferred in multiple I/O requests by the HDF5 library. For chunked storage layouts, each chunk with data is transferred independently and can be placed anywhere on the physical storage media. Only chunks with data in the hyperslab of interest will be transferred.**

Figure 8 depicts the physical layout of the bytes in the hyperslab and array variable on disk for each of the storage layouts tested (not to scale). With contiguous storage, the bytes of interest are not contiguous on disk but are spread throughout the array variable, which is read in its entirety. When chunked storage is used, the number of bytes read corresponds exactly to the number of bytes in the hyperslab. While this hyperslab demonstrates an extreme example of a non-contiguous access pattern, it highlights the data transfer benefits that chunked storage layouts have to offer.

In summary, this example illustrates how the chunked storage layout available with netCDF-4 can tremendously improve the I/O performance for applications with non-contiguous access patterns. The key to improving the performance is to make the shape of the chunk similar to the shape of hyperslab selection. That said, the shapes need not be an exact match, as they were in this example, to see performance benefits.

For small problem sizes, system caching can often greatly reduce the performance penalty caused by non-contiguous access patterns with a contiguous storage layout. Since chunked storage may involve extra overhead, applications with only a small degree of discontinuity in their access patterns may not benefit from the chunked storage layout. Usually, the more discontinuous the access pattern, the greater the performance gains that can be realized with a chunked storage layout. For cases where the hyperslab selections vary depending on the application accessing the variable, using a chunk size that is a compromise between the various hyperslab sizes may be a good option.

## 4.2. Data Compression

NetCDF-4 supports in-memory DEFLATE [8] data compression through the HDF5 library. The compression algorithm performs lossless compression with a range of compression levels supporting different size/speed tradeoffs. For data that can be compressed well with DEFLATE compression, this feature can result in a much smaller file size, with a potential reduction in data transfer time for the compressed data. Compression is specified when an array variable (dataset) is created. All compress/uncompress operations are handled automatically by the HDF5 library, and are transparent to the application.

For this report, DEFLATE compression was applied to netCDF radar sample data obtained from Unidata. Three netCDF files—*tile1*, *tile2*, and *tile4*—were used in the benchmarks. Each file has 12 variables and each variable has 1501×2001 elements of type short. The variables include reflectivity fields, severe hail index (SHI), probability of severe hail (POSH), maximum expected hail size (MEHS), and others. Each variable is about 6 MB in size, for a file size of about 72 MB.

Chunked storage layout is required to support data compression. The chunk size was set to be equal to the size of each variable (1501×2001). Ten runs were made on the AMD-based machine. *ncsync* and *fsync* calls were made after each write to flush the data to disk, and *drop\_caches* was called prior to each read run to clear the system cache. For the performance tests with the radar sample data, the best and worst cases for each configuration were dropped, and the reported results are the average of the eight remaining executions.

Radar Sample Data File	Compression Ratio
tile1	21
tile2	14
tile4	23

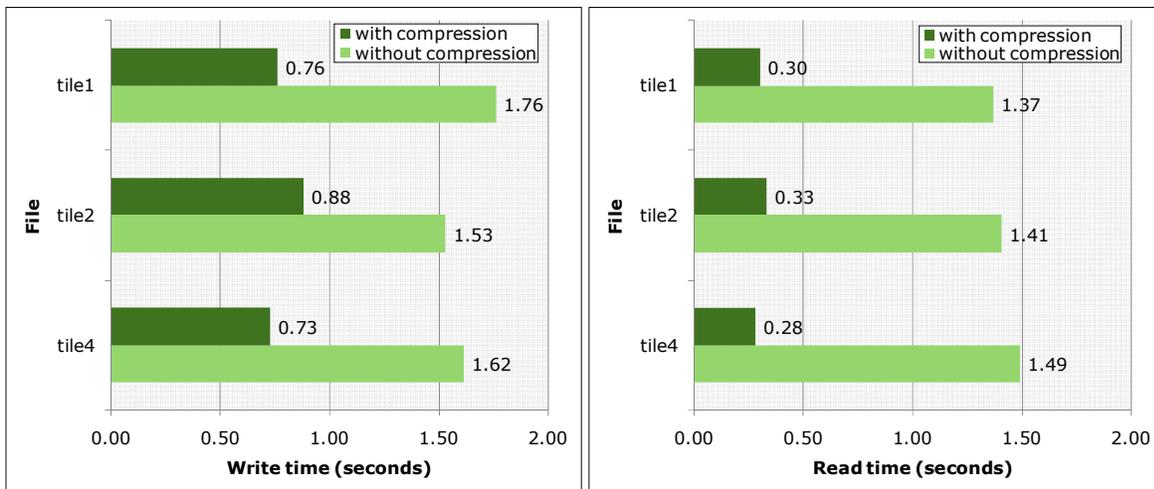
**Table 5 : Compression ratios for netCDF radar sample data files with DEFLATE compression level 1.**

Table 5 shows the compression ratios achieved using DEFLATE compression level 1. While higher compression levels further reduce the data size, they do so at the cost of extra processing

time. For the radar sample data files, the higher compression levels did not result in substantially better degrees of compression, and were not worth the extra compute overhead. Different data will exhibit different compression characteristics.

Figure 9 shows the elapsed wall clock time to write and read the variables in each of the three netCDF radar sample data files with and without compression. For the tests with the uncompressed files, the variations in the elapsed time to read or write the files is due to system fluctuations, as the file sizes, variable sizes, and operations performed are identical.

For the tests with DEFLATE, the reported time includes the compression (for write) and decompression (for read) operations that are automatically applied by the HDF5 library. Even with these operations, there was a considerable performance benefit when the DEFLATE compression was used. The reduced file size lowered the data transfer time by more than enough to offset the compute time spent on compression and decompression. The netCDF-4 / HDF5 compression option offers both space savings and improved read/write performance in these tests.



**Figure 9: NetCDF-4 elapsed time to write and read radar sample data with DEFLATE compression level 1 and without compression; system cache cleared and ncsync/fsync used, AMD-based machine.**

## 5. Performance Pitfalls with NetCDF-4

The previous section presented cases where netCDF-4 can be used to improve I/O performance. Advice on how to avoid poor performance with netCDF-4 is the focus of this section.

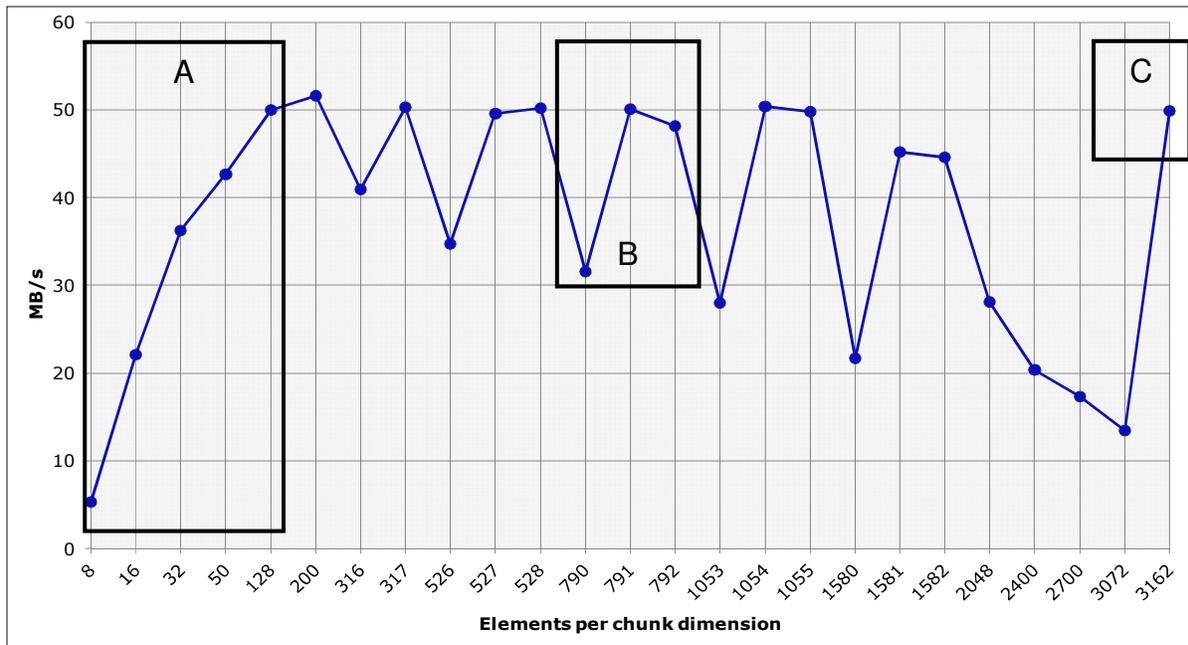
### 5.1. Chunk Size

As shown in the previous section, chunked storage in HDF5 is a powerful technique that can dramatically improve I/O performance for some applications. However, poor choices when configuring chunked storage can lead to unexpectedly bad performance. A common pitfall is the choice of chunk sizes that are not appropriate for an application’s variables and access patterns. A comprehensive study on chunk size specification is beyond the scope of this paper. As an alternative to a rigorous presentation, experiments with various chunk sizes are presented to motivate guidelines for chunk size selection. The experiments focus on how chunk size affects write rates and file sizes.

### 5.1.1. Chunk Size Experiments

Consider a  $3162 \times 3162$  two-dimensional integer array variable with a total size just under 40 MB. For the experiments, the entire array variable was covered by a single hyperslab and written with one *nc\_put\_vara* call. Twenty-five cases, corresponding to twenty-five different chunk sizes, were tested. The chunks used were all square (the number of elements in each chunk dimension was the same), and ranged in size from  $8 \times 8$  to  $3162 \times 3162$ . Note that the chunk sizes do not increase uniformly across the range, but were chosen to highlight important selection criteria. Ten runs for each of the twenty-five cases were made on the AMD-based system with *ncsync/flush* called to flush data to disk. The best and worst runs for each case were dropped, and the reported results are the average of the eight remaining executions.

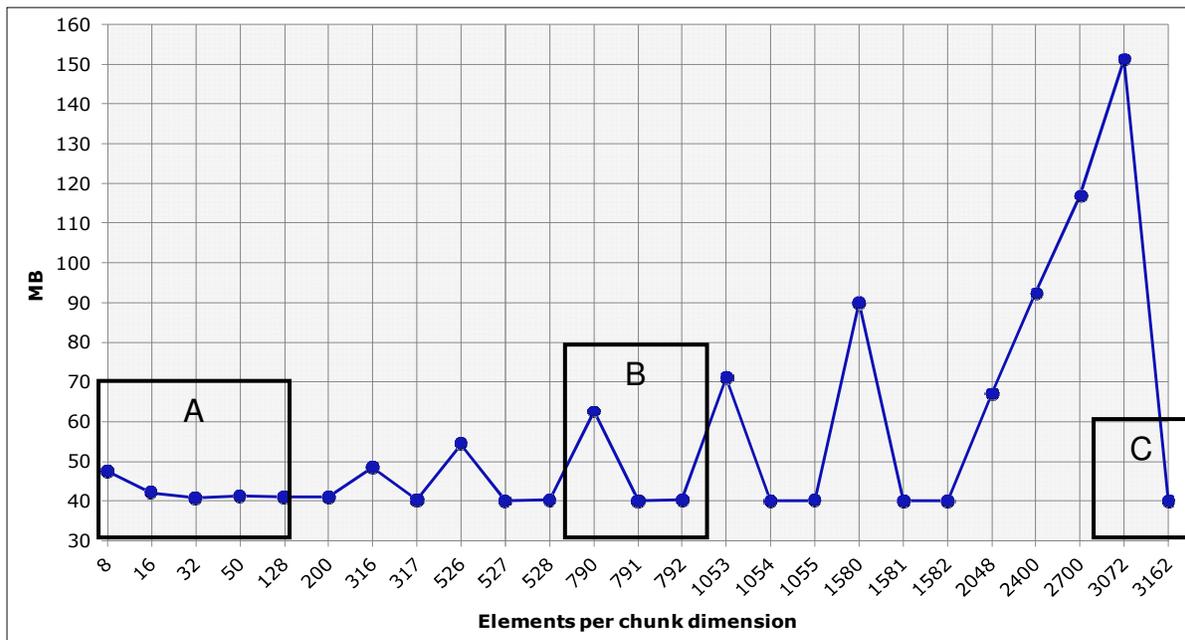
Figures 10 and 11 show the write rate (size of the array variable divided by elapsed wall clock time) and the size of the file created for the twenty-five different chunk sizes. Write rates ranged from 5.29 to 51.59 MB/s, and file sizes ranged from 40 to 151 MB. Closer investigation will give insights into this extreme variability in both performance and file size.



**Figure 10: Write rate for  $3162 \times 3162$  integer array with chunked storage layout and various chunk sizes; *ncsync/flush* used, AMD-based machine. Regions A, B, and C are discussed in the text.**

Consider first the experiments with the smallest chunks ranging in size from  $8 \times 8$  to  $128 \times 128$ , corresponding to the region marked “A” in Figures 10 and 11. In Figure 10 it is clear that the write rate was slowest with the smallest chunk size. The write rate increased substantially until the chunk size reached  $128 \times 128$ . Since chunks are written individually, the smaller chunk sizes mean less data is being written at a time, more writes occur, and overhead is higher. The overhead is especially apparent when system caching is not used and each write flushes the data to disk. Note also that the file size was about 20% larger than the size of the array data when the  $8 \times 8$  chunk size was used. Looking at Figure 11, the file size for the experiments in region A was smallest for the  $32 \times 32$  chunk size. Even though the  $50 \times 50$  and  $128 \times 128$  chunk sizes yield

slightly larger files, they resulted in better write rates because fewer writes occurred and there was more data transferred with each write. A closer examination of file sizes follows.



**Figure 11: Size of the file created for 3162x3162 integer array with chunked storage layout and various chunk sizes; ncsync/fsync used, AMD-based machine. Regions A, B, and C are discussed in the text.**

For the chunk sizes from 316x316 to 3162x3162 there was considerable fluctuation in both the write rate and the file size. Looking carefully at Figures 10 and 11, chunk sizes that exhibited dips in the write rate also showed peaks in the file size. This correlation indicates that the decreased write rate (from the application’s perspective) was likely due to more bytes actually being written to disk. Recall that the reported write rate is calculated by dividing the size of the array variable by the wall clock time that elapsed while writing the array variable to disk. Also recall that with chunked storage all chunks for a given variable (dataset) are the same size, and chunks are written to disk independently.

Table 6 provides details for the three experiments that fall in the region marked “B” in Figures 10 and 11. The experiments have been named based on the chunk size used. For example, the experiment with chunk size 790x790 is called E790.

Experiment	Chunk Size (nxn)	Write rate (MB/s)	File Size (MB)	3162/n	[3162/n]
E790	790x790	31.52	62.42	4.0025	5
E791	791x791	50.04	40.05	3.9975	4
E792	792x792	48.17	40.15	3.9924	4

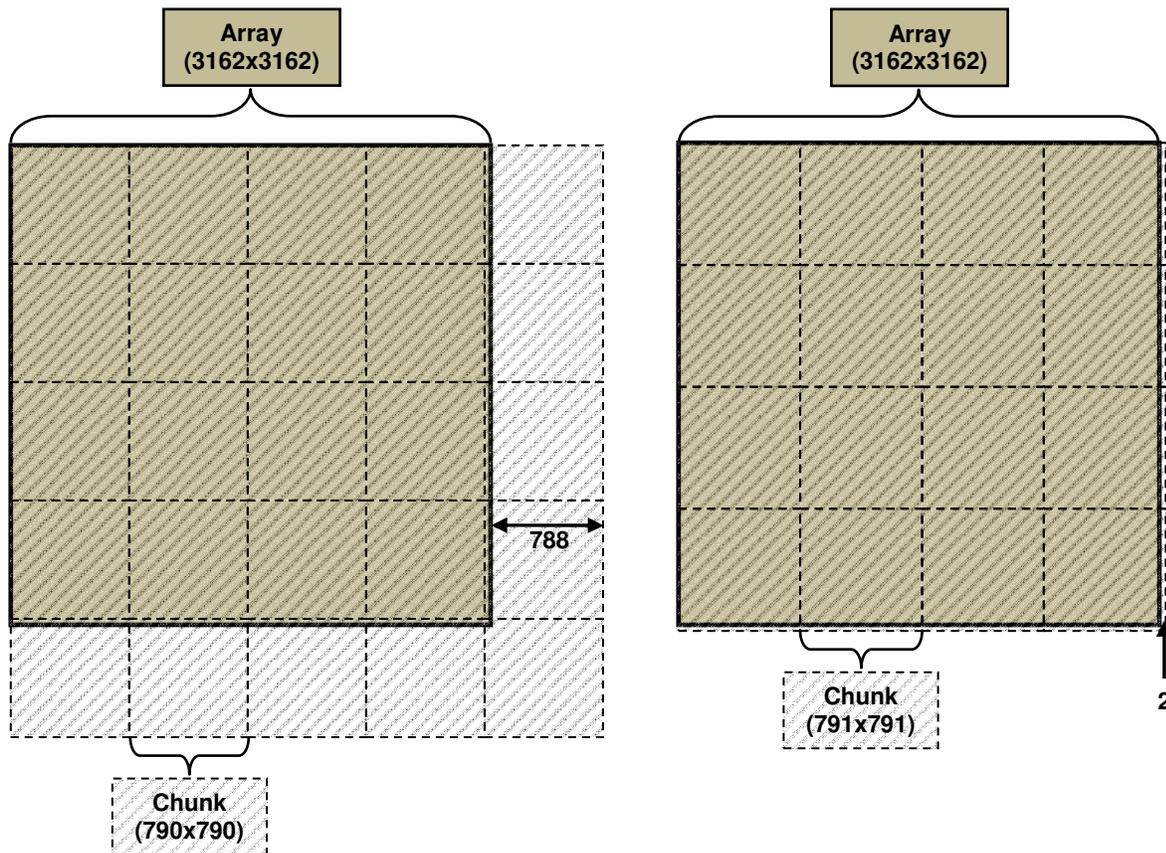
**Table 6: Details for experiments appearing in Region B of Figures 10 and 11.**

Table 6 shows the write rate for E790 was about 32 MB/s while the write rates for E791 and E792 were closer to 50 MB/s. The netCDF-4 file written by E790 was 62 MB—more than 50% larger than the original array variable size of 40 MB. In contrast, the files for E791 and E792

were very close to size of the original array. The three experiments highlight how a very minor change in chunk size can result in very major shifts in I/O rates and file sizes.

Why does the file size change so dramatically with a small change in chunk size? The 3162x3162 array variable must be totally covered by equally-sized chunks. Since both the array and the chunks are square, a single dimension can be considered when computing coverage requirements. 3162 divided by 790 is 4.0025, meaning that it will take five “chunks” of 790 elements to cover a variable with 3162 elements. The final two columns in Table 6 summarize the coverage calculations for the three experiments. For E790 it takes twenty-five (5\*5=25) chunks to cover the array variable, while for E791 and E792 only sixteen (4\*4=16) chunks are required.

Figure 12 illustrates the chunk coverage of the array variable for E790 and E791. For E790 there are nine chunks containing very little array data and mostly unused space. Since chunks are written in their entirety, this unused space ( ( 25\*790\*790 – 3162\*3162 ) \* 4 bytes ) accounts for the 22 MB of “extra” file space noted earlier. In contrast, for E791 there is very little unused space ( ( 16\*791\*791 – 3162\*3162 ) \* 4 ), about 50KB.



**Figure 12: Coverage of 3162x3162 array variable with 790x790 and 791x791 chunks corresponding to Experiments E790 and E791 in Table 6.**

With an understanding the actual volume of data written, it is not surprising that the application’s observed write rate for E791 was over 1.5 times better than the write rate for E790. The performance for E792 was similar to that of E791, with a marginal increase in file size. The

difference between E790 and E792 shows that “off by one” in chunk size selection can result in dramatically different results depending on the direction you are “off by”.

Finally, consider the experiment in the region marked “C” in Figures 10 and 11. In this experiment a single chunk of size 3162×3162 was used. The write rate was 49.87 MB/s and the file size was only slightly larger than the size of the array variable, with the extra 1281 bytes accommodating the HDF5 metadata. The single chunk experiment offers very close to top performance combined with minimal disk space consumption.

### 5.1.2. Chunk Size Selection Guidelines

Four important guidelines for the selection of good chunk sizes can be drawn from the experiments.

1. Always avoid using a small chunk size. In the 2D array case shown, the performance was extremely degraded when the chunk size was smaller than 32×32. Because the overhead of chunk management is a fixed amount per chunk, regardless of chunk size, and because it is generally more efficient to write larger blocks of data to the disk, the performance will be poor when small chunks are used.
2. If the system where the application is running has sufficient memory and the access pattern is contiguous or nearly contiguous, using a single chunk sized to exactly match the array variable can be an excellent choice. While this is a good general guideline, some applications may achieve better performance with a more complex sizing strategy.
3. If there are reasons to choose a chunk size that is smaller than the size of the array variable, set the number of elements in a given chunk dimension ( $n$ ) to be the ceiling of the number of elements in that dimension of the array variable ( $d$ ) divided by a natural number  $N > 1$ . That is, set  $n = \lceil d / N \rceil$ . For the example shown in Figure 12, the good chunk size had 791 elements per dimension and  $791 = \lceil 3162 / 4 \rceil$ . Using a chunk size slightly larger than this value is also acceptable.
4. If there are reasons to choose a chunk size that is smaller than the size of the array variable, avoid setting the number of elements in a given chunk dimension ( $n$ ) to be the floor of the number of elements in that dimension of the array variable ( $d$ ) divided by a natural number  $N > 1$ . That is, do not set  $n = \lfloor d / N \rfloor$ . For the example shown in Figure 12, the bad chunk size had 790 elements per dimension and  $790 = \lfloor 3162 / 4 \rfloor$ . Chunk sizes slightly smaller than this value should also be avoided.

## 5.2. Number of Hyperslabs

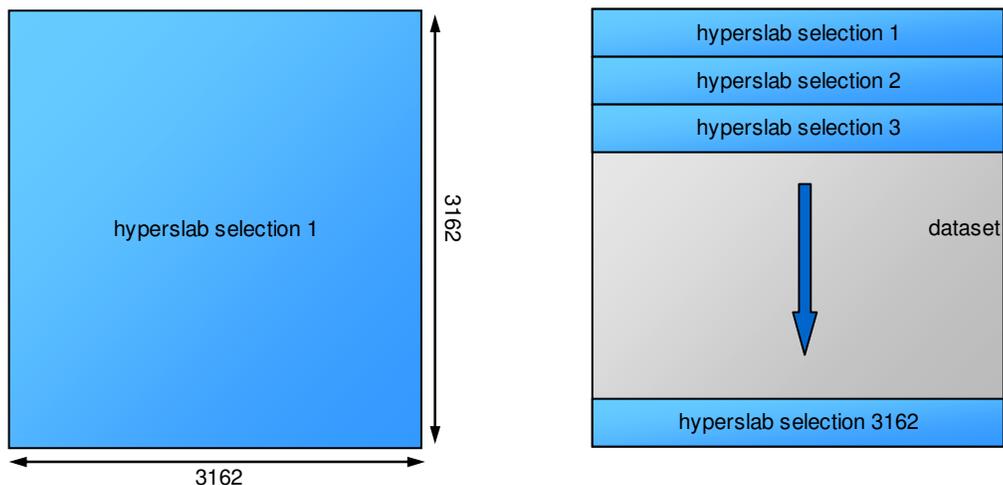
While the use of hyperslabs, discussed in Section 4.1.1, offers an intuitive way for applications to write or read subsets of an array variable, the use of many small hyperslabs may cause poor performance. Once again, experimental results are used to demonstrate the possible pitfall.

### 5.2.1. Hyperslab Experiments

As with the chunk size experiments in the previous section, the hyperslab experiments also used a 3162×3162 two-dimensional integer array variable with a total size slightly less than 40 MB. Runs were made on the Intel-based system with the default system caching, using both chunked

and contiguous storage layouts. For the chunked storage, a chunk size of  $3162 \times 3162$  was specified. Ten runs were made for each unique experiment and the reported results are the average of the eight runs remaining after the best and worst runs were dropped.

Figure 13 depicts the two hyperslab selection configurations used in the experiments. Table 7 summarizes the write and read rates for the two configurations with contiguous and chunked storage layouts.



**Figure 13: Hyperslab selection configurations. One large  $3162 \times 3162$  hyperslab covering the entire array variable and 3162 hyperslabs, each of size  $1 \times 3162$ .**

Storage Layout	Write Rate (MB/s)		Read Rate (MB/s)	
	Single Hyperslab	3162 Hyperslabs	Single Hyperslab	3162 Hyperslabs
Contiguous	288.59	111.85	412.78	268.05
Chunked	133.88	82.27	421.43	232.16

**Table 7: NetCDF-4 write and read performance for  $3162 \times 3162$  integer array variable accessed using a single large  $3162 \times 3162$  hyperslab selection and using 3162 small  $1 \times 3162$  hyperslab selections. Chunk size was  $3162 \times 3162$  for chunked storage layouts. Default system caching on Intel-based machine.**

For both writes and reads with contiguous and chunked storage, the performance was lower when the 3162 small hyperslabs were used. These experiments were also conducted using the HDF5 library directly, rather than going through netCDF-4. When HDF5 was called directly, the performance for the small hyperslabs was noticeably better.

Using Quantify [9], The HDF Group found there were 3162 calls within the netCDF-4 library to the routines *get\_property\_internal* and *open\_var\_grp\_cached* when the small hyperslabs were used. Investigating further, *get\_property\_internal* and *open\_var\_grp\_cached* are wrappers for the HDF5 library calls *H5Pcreate* and *H5Dopen*. Since all of the accesses are for the same array variable, just different subsets of it, it is not necessary to call *H5Pcreate* and *H5Dopen* for each of the hyperslab selections.

Taking advantage of its understanding of the HDF5 library, The HDF Group modified the netCDF-4 source code to reuse HDF5 descriptors when one variable is accessed repeatedly. Table 8 shows the write and read rates measured for the 3162 small hyperslab selections after the modification. Improvements were seen for the small hyperslabs in all configurations tested, with the most dramatic increases seen for reads. After the modifications, the small hyperslab selections performed better than the single large hyperslab for reads using the contiguous storage layout. This may be due to caching within HDF-5, but that has not been verified.

Storage Layout	Write Rate (MB/s) 3162 Hyperslabs	Read Rate (MB/s) 3162 Hyperslabs
Contiguous	131.26	504.17
Chunked	104.97	421.81

**Table 8: Modified NetCDF-4 write and read performance for 3162x3162 integer array variable accessed using 3162 small 1x3162 hyperslab selections. Chunk size was 3162x3162 for chunked storage layouts. Default system caching on Intel-based machine.**

### 5.2.2. Hyperslab Selection Guidelines

With the current release of netCDF-4, the use of many hyperslabs to write or read an array variable should be avoided. Performance under these conditions may improve in future releases, as netCDF-4 is optimized to reduce the number of calls to HDF5 when multiple reads or writes are made to the same array variable (dataset). Even after those modifications, the use of larger hyperslabs may still be preferred, especially for writes.

## 6. Conclusions

A variety of benchmark results and examples were presented in this report to help users of netCDF gain a better understanding of netCDF-4's performance characteristics.

For cases where the features supported by the chunked storage layout are not needed, the use of contiguous storage with netCDF-4 is usually a better choice. When contiguous storage is used, the performance of netCDF-4 is comparable to that of netCDF-3 overall, with considerable variation in some cases depending on the data size and access operation (write or read). Table 2 summarizes the benchmark results for the netCDF-3 and netCDF-4 performance comparisons.

For highly non-contiguous access patterns, a chunked storage layout can dramatically improve performance. See Figure 7 for details.

For some applications, the use of netCDF-4's in-memory compression feature, which requires chunked storage, can not only reduce the data storage size but also reduce the overall time to write and read data. See Figure 9 for details.

When chunked storage is used, it is critical that a good chunk size be chosen. Small chunks should be avoided, and care should be taken so that the chunk size selected does not result in large amounts of unused space in the file. Figures 10 and 11 show the fluctuations that can occur in write rate and file size for different chunk sizes. Section 5.1.2 provides guidelines for good chunk size selection.

The use of many hyperslabs with a single array variable can cause poor performance in the initial release of netCDF-4, and should be avoided. Section 5.2.1 discusses this issue.

Several areas for possible performance improvements were identified in the course of the benchmark tests. They include the default storage allocation policy with chunked storage, the cache size for large datasets with chunked storage, and unnecessary calls in the netCDF-4 library when multiple hyperslabs are used to access a single array variable. The HDF Group will be working with Unidata to address these issues in future releases. In addition, further tests will be conducted to identify other opportunities for optimization.

## Acknowledgements

This report is based on work supported in part by a Cooperative Agreement with the National Aeronautics and Space Administration (NASA) Earth Science Data Information System Project (ESDIS) under NASA grant NNX06AC83A, and by the Department of Energy (DOE) Sandia National Laboratory under contract number 605603, “HDF support for Advanced Simulation and Computing (ASC)”. Any opinions, findings, conclusions, or recommendations expressed in this report are those of the authors and do not necessarily reflect the views of NASA or DOE.

We thank Ed Hartnett and Russ Rew, from the University Corporation for Atmospheric Research (UCAR) Unidata program, who provided the initial benchmark code and radar data. We also thank Elena Pourmal, Quincey Koziol, Albert Cheng, and Mike Folk, colleagues at The HDF Group, who provided valuable suggestions and assistance during the course of this work. We also thank Dan Marinelli, NASA ESDIS manager, for his support in making this study possible.

## References

- [1] netCDF4: <http://www.unidata.ucar.edu/software/netcdf/netcdf-4/>
- [2] netCDF: <http://www.unidata.ucar.edu/software/netcdf/>
- [3] HDF5: <http://www.hdfgroup.org/HDF5/>
- [4] NASA Earth Science Technology Office: <http://esto.nasa.gov>
- [5] Unidata: <http://www.unidata.ucar.edu/>
- [6] The HDF Group: <http://www.hdfgroup.org/>
- [7] /proc/sys/vm/drop\_caches: [http://www.linuxinsight.com/proc\\_sys\\_vm\\_drop\\_caches.html](http://www.linuxinsight.com/proc_sys_vm_drop_caches.html) and <http://linux.die.net/man/5/proc>
- [8] DEFLATE: RFC 1951 DEFLATE Compressed Data Format Specification version 1.3 <http://tools.ietf.org/html/rfc1951>
- [9] IBM Rational Quantify: <http://www.ibm.com/software/rational/>