Chapter 14

HDF Performance Issues

14.1 Chapter Overview and Introduction

This chapter describes many of the concepts the HDF user should understand to gain better performance from their applications that use the HDF library. It also covers many of the ways in which HDF can be used to cause impaired performance and methods for correcting these problems.

As stated earlier in this manual, HDF has been designed to be very general-purpose, and it has been used in many different applications involving scientific data. Each application has its own set of software and hardware resource constraints that will affect performance in a different way, and to a different extent, from the resource constraints in other applications.

Therefore, it is impossible to outline *all* of the performance issues that may relate to a particular application of HDF. However, this chapter should give the reader sufficient knowledge of the most common performance issues encountered by the HDF Group. This knowledge should enable the reader to explore different ways of storing data on their native platforms for the purpose of increasing library performance.

Future versions of this chapter will include additional possibilities of performance enhancement as they are discovered.

14.2 Examples of HDF Performance Enhancement

In this section, four pairs of HDF object models along with their C implementations will be presented. Each pair will illustrate a specific aspect of HDF library performance as it relates to scientific data sets. They will be employed here as general pointers on how to model scientific data sets for optimized performance.

In developing and testing these examples, the Sun Solaris OS version supported by HDF version 4.1 release 1 was used. Version 2.0 of the Quantify performance profiler was used to measure the relative differences in library performance between the SDS models in each pair. It should be noted that, while the examples reliably reflect which SDS configurations result in better performance, the specifics of how much performance will be improved depend on many factors such as OS configuration, compiler used and profiler used. Therefore, any specific measurements of performance mentioned in the chapter should be interpreted only as general indicators.

The C programs that were used as the basis of this section can be obtained from the HDF ftp server at hdf.ncsa.uiuc.edu in the /pub/dist/HDF/HDF4.1r1/Performance directory. These are provided in the event the reader wishes to verify or modify these examples on their own system.

The reader should keep in mind that the following examples have been designed for illustrative purposes only, and should not be considered as real-world examples. It is expected that the reader will apply the library performance concepts covered by these examples to their specific usage of the HDF library.

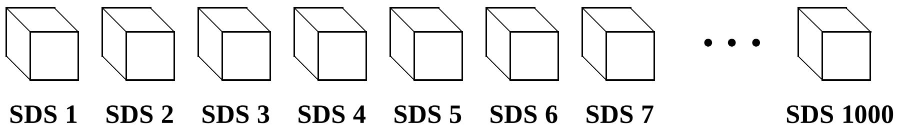
14.2.1 One Large SDS Versus Several Smaller SDSs

The scientific data set is an example of what in HDF parlance is referred to as a *primary object*. The primary objects accessed and manipulated by the HDF library include, beside scientific data sets, raster images, annotations, vdatas and vgroups. Each primary object has *metadata*, or data describing the data, associated with it. Refer to the *HDF Specifications Manual* for a description of the components of this metadata and how to calculate its size on disk.

An opportunity for performance enhancement can exist when the size of the metadata far exceeds the size of the data described by the metadata. In this situation, more CPU time and disk space will be used to maintain the metadata than the data contained in the SDS. Consolidating the data into fewer, or even one, SDS can increase performance.

To illustrate this, consider 1,000 1 *x* 1 *x* 1 element scientific data sets of 32-bit floating-point numbers. No user-defined dimension, dimension scales or fill values have been defined or created.

FIGURE 14a **1,000 1 *x* 1 *x* 1 Element Scientific Data Sets**



In this example, 1,000 32-bit floating-point numbers are first buffered in-core, then written to the SDS.

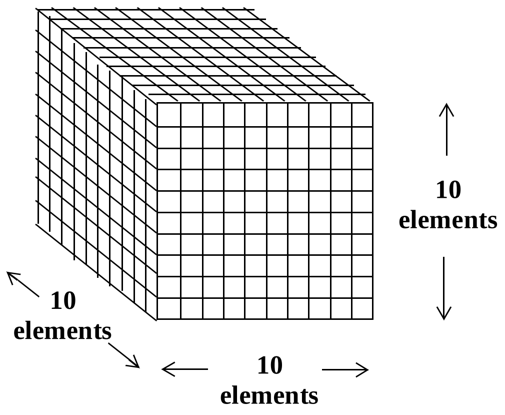
In Table 14A, the results of this operation are reflected in two metrics: the total number of CPU cycles used by the example program, and the size of the HDF file after the write operation.

TABLE 14A **Results of the Write Operation to 1,000 1 *x* 1 *x* 1 Element Scientific Data Sets**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
136,680,037	896,803

Now the 1,000 32-bit floating point numbers that were split into 1,000 SDSs are combined into one 10 *x* 10 *x* 10 element SDS. This is illustrated in the following figure.

FIGURE 14b **One 10 *x* 10 *x* 10 Element Scientific Data Set**



As with the last example, 1,000 32-bit floating-point numbers are first buffered in-core, then written to the SDS. The following table contains the performance metrics of this operation.

TABLE 14B **Results of the Write Operation to One 10 *x* 10 *x* 10 Element Scientific Data Set**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
205,201	7,258

It is apparent from these results that merging the data into one scientific data set results in a substantial increase in I/O efficiency - in this case, a 99.9% reduction in total CPU load. In addition, the size of the HDF file is dramatically reduced by a factor of more than 100, even through the amount of SDS data stored is the same.

The extent to which the data consolidation described in this section should be done is dependent on the specific I/O requirements of the HDF user application.

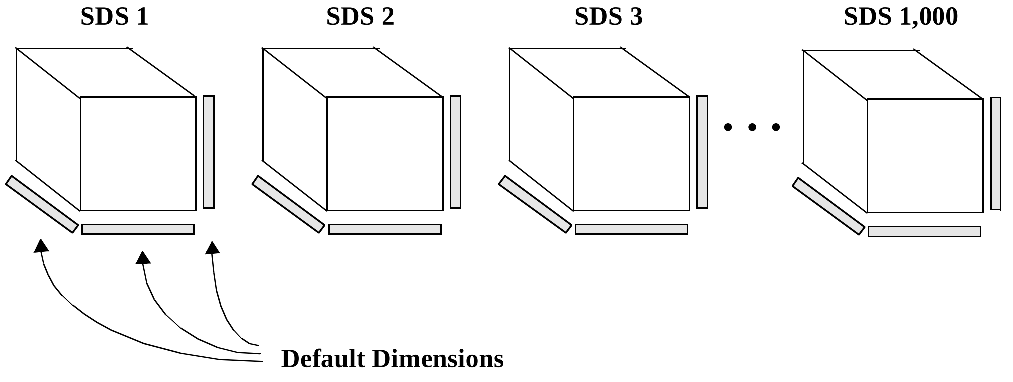
14.2.2 Sharing Dimensions Between Scientific Data Sets

When several scientific data sets have dimensions of the same length, name and data type, they can share these dimensions to reduce storage overhead and CPU cycles in writing out data.

To illustrate this, again consider the example of 1,000 1 *x* 1 *x* 1 scientific data sets of 32-bit floating point numbers. Three dimensions are attached by default to each scientific data set by the HDF library. The HDF library assigns each of these dimensions a default name prefaced by "fakeDim". See Chapter 3 of the *HDF User’s Guide*, titled *Scientific Data Sets (SD SDS)*, for a specific explanation of default dimension naming conventions.

FIGURE 14c

1,000 1 *x* 1 *x* 1 Element Scientific Data Sets



One 32-bit floating point number is written to each scientific data set. The following table lists the performance metrics of this operation.

TABLE 14C

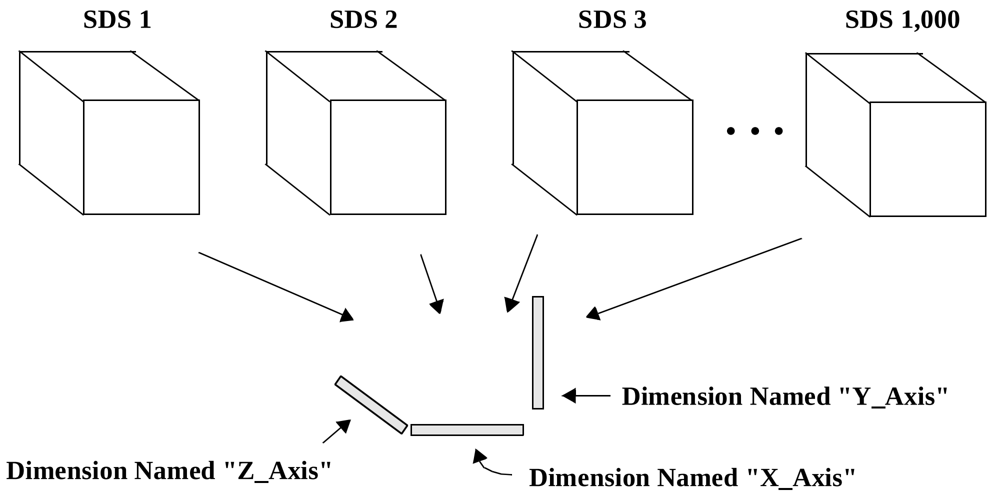
Results of the Write Operation to 1,000 1 *x* 1 *x* 1 Element Scientific Data Sets

Total Number of CPU Cycles	Size of the HDF File (in bytes)
136,680,037	896,803

Now consider the 1,000 SDSs described previously in this section. In this case, the 1,000 SDSs share the program-defined "X_Axis", "Y_Axis" and "Z_Axis" dimensions as illustrated in the following figure.

FIGURE 14d

1,000 1 *x* 1 *x* 1 Element Scientific Data Sets Sharing Dimensions



The performance metrics that result from writing one 32-bit floating-point number to each dataset are in the following table.

TABLE 14D **Results of the Write Operation to 1,000 1 *x* 1 *x* 1 SDSs with Shared Dimensions**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
24,724,384	177,172

A 82% performance improvement in this example program can be seen from the information in this table, due to the fewer write operations involved in writing dimension data to shared dimensions. Also, the HDF file is significantly smaller in this case, due to the smaller amount of dimension data that is written.

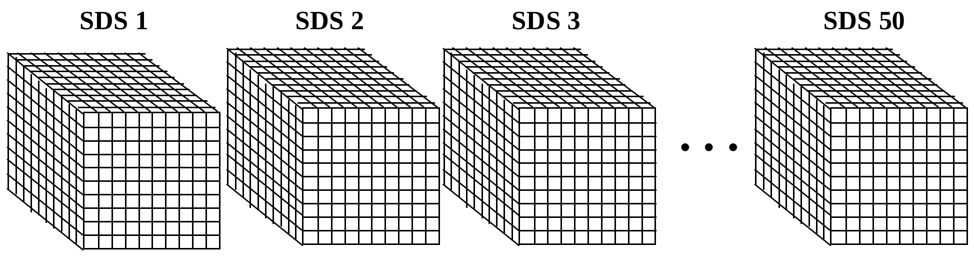
14.2.3 Setting the Fill Mode

When a scientific data set is created, the default action of the HDF library is to fill every element with the default fill value. This action can be disabled, and reenabled once it has been disabled, by a call to the **SDsetfillmode** routine.

The library's default writing of fill values can degrade performance when, after the fill values have been written, every element in the dataset is written to again. This operation involves writing every element in the SDS twice. This section will demonstrate that disabling the initial fill value write operation by calling **SDsetfillmode** can improve library performance.

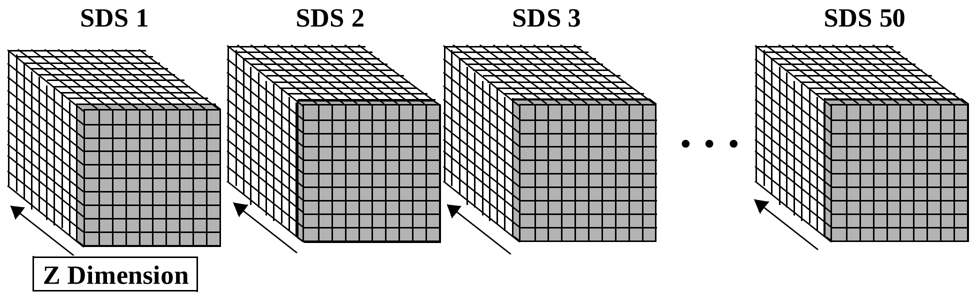
Consider 50 10 *x* 10 *x* 10 scientific data sets of 32-bit floating-point numbers.

FIGURE 14e **50 10 *x* 10 *x* 10 Element Scientific Data Sets**



By default, the fill value is written to every element in all 50 SDSs. The contents of a two-dimensional buffer containing 32-bit floating-point numbers is then written to these datasets. The way these two-dimensional slices are written to the three-dimensional SDSs is illustrated in the following figure. Each slice (represented by each shaded area in the figure) is written along the third dimension of each SDS, or if the dimensions are related to a Cartesian grid, the z-dimension, until the entire SDS is filled.

FIGURE 14f **Writing to the 50 10 *x* 10 *x* 10 Element Scientific Data Sets**



It should be noted that the reason each SDS isn't rewritten to in one write operation is because the HDF library will detect this and automatically disable the initial write of the fill values as a performance-saving measure. Hence, the partial writes in two-dimensional slabs.

The following table shows the number of CPU cycles needed in our tests to perform this write operation with the fill value write enabled. The "Size of the HDF File" metric has been left out of this table, because it won't change substantially regardless of whether the default fill value write operation is enabled.

TABLE 14E **Results of the Write Operation to the 50 10 *x* 10 *x* 10 SDSs with the Fill Value Write Enabled**

Total Number of CPU Cycles
584,956,078

The following table shows the number of CPU cycles needed to perform the same write operation with the fill value write disabled.

TABLE 14F **Results of the Write Operation to the 50 SDSs with the Fill Value Write Disabled**

Total Number of CPU Cycles
390,015,933

The information in these tables demonstrate that eliminating the I/O overhead of the default fill value write operation when an entire SDS is rewritten to results in a substantial reduction of the CPU cycles needed to perform the operation - in this case, a reduction of 33%.

14.2.4 Disabling "Fake" Dimension Scale Values in Large One-Dimensional Scientific Data Sets

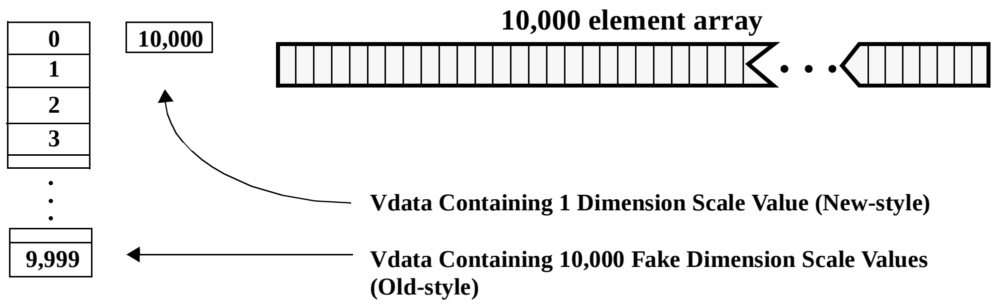
In versions 4.0 and earlier of the HDF library, dimension scales were represented by a vgroup containing a vdata. This vdata consisted of as many records as there are elements along the dimension. Each record contained one number which represented each value along the dimension scale, and these values are referred to as "fake" dimension scale values.

In HDF version 4.0 a new representation of the dimension scale was implemented alongside the old one - a vdata containing only one value representing the total number of values in the dimension scale. In version 4.1 release 1, this representation was made the default. A "compatible" mode is also supported where both the older and newer representations of the dimension scale are written to file.

In the earlier representation, a substantial amount of I/O overhead is involved in writing the fake dimension scale values into the vdata. When one of the dimensions of the SDS array is very large, performance can be improved, and the size of the HDF file can be reduced, if the old representation of dimension scales is disabled by a call to the **SDsetdimval_comp** routine. The examples in this section will illustrate this.

First, consider one 10,000 element array of 32-bit floating point numbers, as shown in the following figure. Both the new and old dimension scale representations are enabled by the library.

FIGURE 14g **One 10,000 Element Scientific Data Set with Old- and New-Style Dimension Scales**


10,000 element array
0
1
2
3
9,999
10,000
Vdata Containing 1 Dimension Scale Value (New-style)
Vdata Containing 10,000 Fake Dimension Scale Values (Old-style)

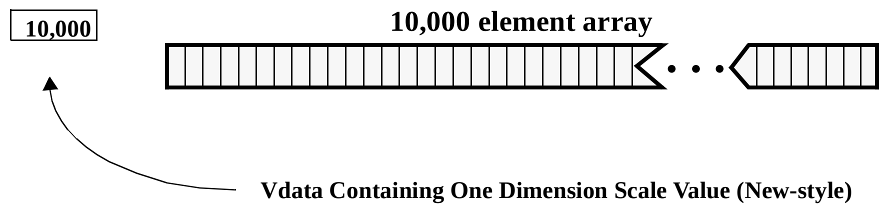
10,000 32-bit floating-point numbers are buffered in-core, then written to the scientific data set. In addition, 10,000 integers are written to the SDS as dimension scale values. The following table contains the results of this operation from our tests.

TABLE 14G **Results of the SDS Write Operation with the New and Old Dimension Scales**

Total Number of CPU Cycles	Size of the HDF File (in bytes)
439,428	82,784

Now consider the same SDS with the fake dimension scale values disabled. The following figure illustrates this.

FIGURE 14h **One 10,000 Element Scientific Data Set With the Old-Style Dimension Scale Disabled**



The following table contains the performance metrics of this write operation.

TABLE 14H **Results of the SDS Write Operation With Only the New Dimension Scale**

Total Number of CPU Cycles	Size of the HDF File
318,696	42,720

The old-style dimension scale is not written to the HDF file, which results in the size of the file being reduced by nearly 50%. There is also a marginal reduction in the total number of CPU cycles.

14.3 Data Chunking

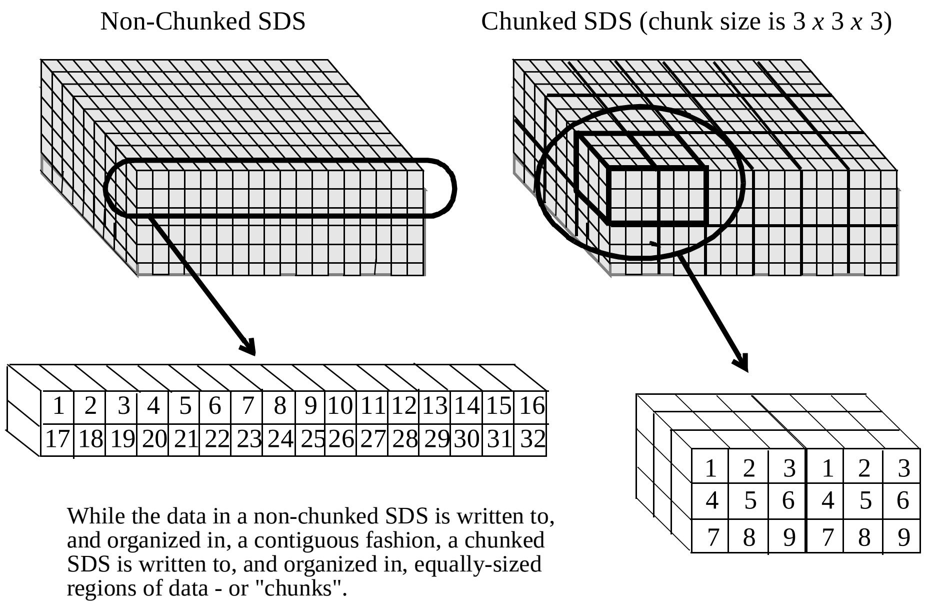
14.3.1 What is Data Chunking?

Data chunking is a method of organizing data within an SDS where data is stored in "chunks" of a predefined size, rather than contiguously by array element. Its two-dimensional instance is sometimes referred to as "data tiling". Data chunking is generally beneficial to I/O performance in very large arrays - i.e., arrays with thousands of rows and columns.

If correctly applied, data chunking may reduce the number of seeks through the SDS data array to find the data to be read or written, thereby improving I/O performance. However, it should be remembered that data chunking, if incorrectly applied, can significantly *reduce* the performance of reading and/or writing to an SDS. Knowledge of how chunked SDSs are created and accessed and application-specific knowledge of how data is to be read from the chunked SDSs are necessary in avoiding situations where data chunking works against the goal of I/O performance optimization.

The following figure illustrates the difference between a non-chunked SDS and a chunked SDS.

FIGURE 14i **Comparison Between Chunked and Non-Chunked Scientific Data Sets**


Non-Chunked SDS
Chunked SDS (chunk size is 3 x 3 x 3)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
1 2 3 1 2 3
4 5 6 4 5 6
7 8 9 7 8 9
While the data in a non-chunked SDS is written to, and organized in, a contiguous fashion, a chunked SDS is written to, and organized in, equally-sized regions of data - or "chunks".

14.3.2 Writing Concerns and Reading Concerns in Chunking

There are issues in working with chunks that are related to the reading process and others that are related to the writing process.

Specifically, the issues that affect the process of reading from chunked SDSs are

- Compression
- Subsetting
- Chunk sizing
- Chunk cache sizing

The issues that affect the process of writing to chunked SDSs are

- Compression
- Chunk cache sizing

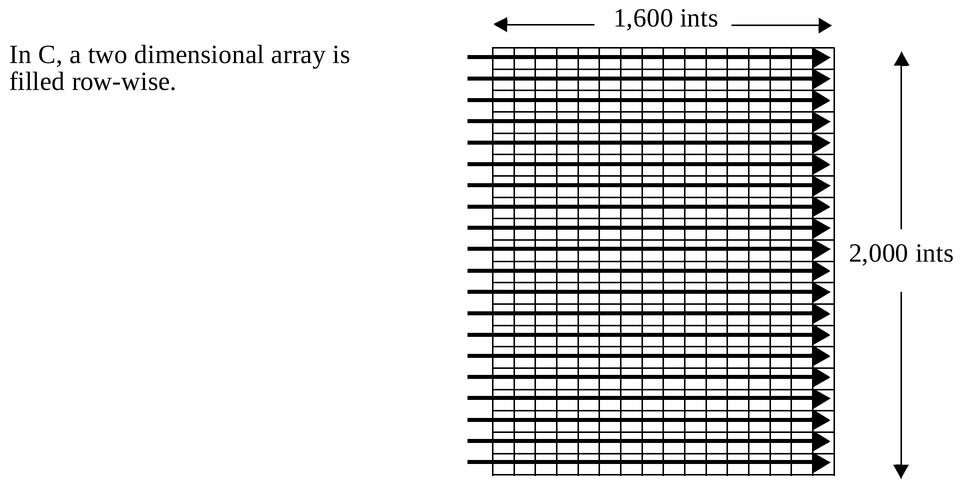
14.3.3 Chunking without Compression

Accessing Subsets According to Storage Order

The main consideration to keep in mind when subsetting from chunked and non-chunked SDSs is that if the subset can be accessed in the same order as it was stored, subsetting will be efficient. If not, subsetting may result in less-than-optimal performance considering the number of elements to be accessed.

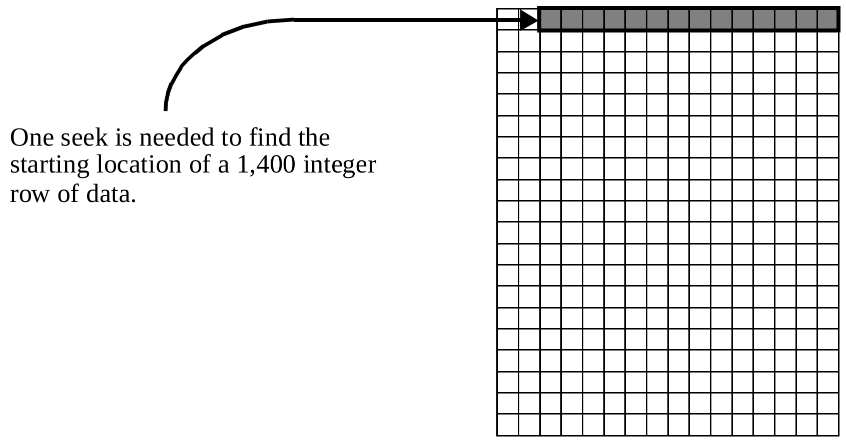
To illustrate this, the instance of subsetting in non-chunked SDSs will first be described. Consider the example of a non-chunked, two-dimensional, 2,000 *x* 1,600 SDS array of integer data. The following figure shows how this array is filled with data in a row-wise fashion. (Each square in the array shown represents 100 *x* 100 integers.)

FIGURE 14j **Filling a Two-Dimensional Array With Data Using Row-Major Ordering**



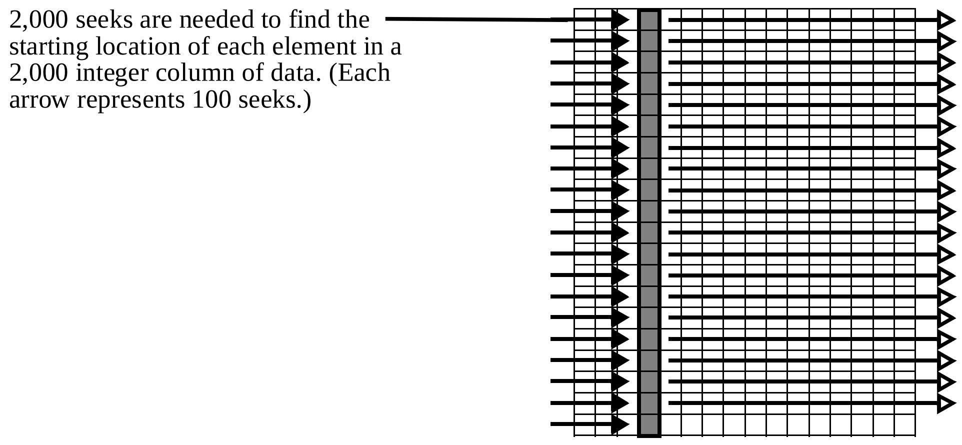
The most efficient way an application can read a row of data, or a portion of a row, from this array, is a contiguous, row-wise read of array elements. This is because this is the way the data was originally written to the array. Only one seek is needed to perform this. (See Figure 14k.)

FIGURE 14k **Number of Seeks Needed to Access a Row of Data in a Non-Chunked SDS**


One seek is needed to find the starting location of a 1,400 integer row of data.

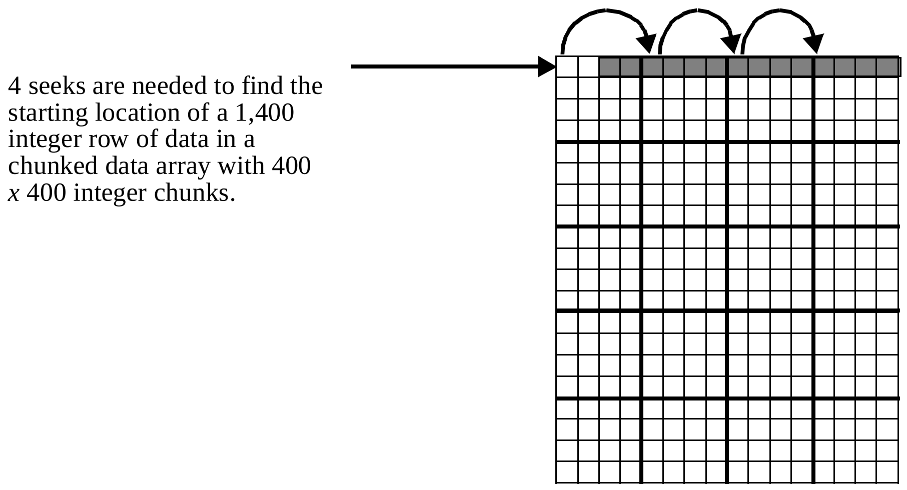
If the subset of data to be read from this array is one 2,000 integer *column*, then 2,000 seeks will be required to complete the operation. This is the most inefficient method of reading this subset as nearly all of the array locations will be accessed in the process of seeking to a relatively small number of target locations.

FIGURE 14l **Number of Seeks Needed to Access a Column of Data in a Non-Chunked SDS**


2,000 seeks are needed to find the starting location of each element in a 2,000 integer column of data. (Each arrow represents 100 seeks.)

Now suppose this SDS is chunked, and the chunk size is 400 *x* 400 integers. A read of the aforementioned row is performed. In this case, four seeks are needed to read all of the chunks that contain the target locations. This is less efficient than the one seek needed in the non-chunked SDS.

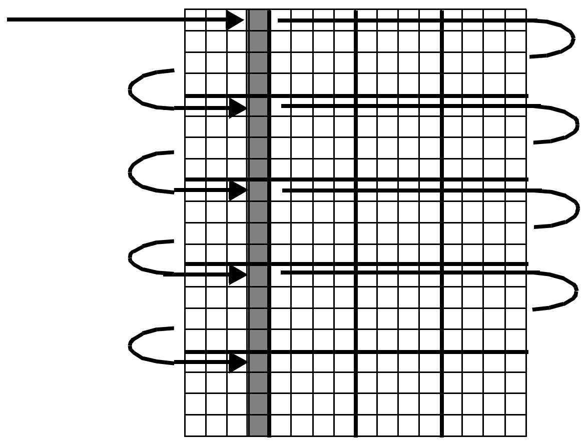
FIGURE 14m **Number of Seeks Needed to Access a Row of Data in a Chunked SDS**


4 seeks are needed to find the starting location of a 1,400 integer row of data in a chunked data array with 400 *x* 400 integer chunks.

To read the aforementioned column of data, five chunks must be read into memory in order to access the 2,000 locations of the subset. Therefore, five seeks to the starting location of each of these chunks are necessary to complete the read operation - far fewer than the 2,000 needed in the non-chunked SDS.

FIGURE 14n **Number of Seeks Needed to Access a Column of Data in a Chunked SDS**

5 seeks are needed to find the starting location of a 2,000 integer column of data in a chunked data array with 400 *x* 400 integer chunks. (Each arrow represents one seek.)



These examples show that, in many cases, chunking can be used to reduce the I/O overhead of subsetting, but in certain cases, chunking can impair I/O performance.

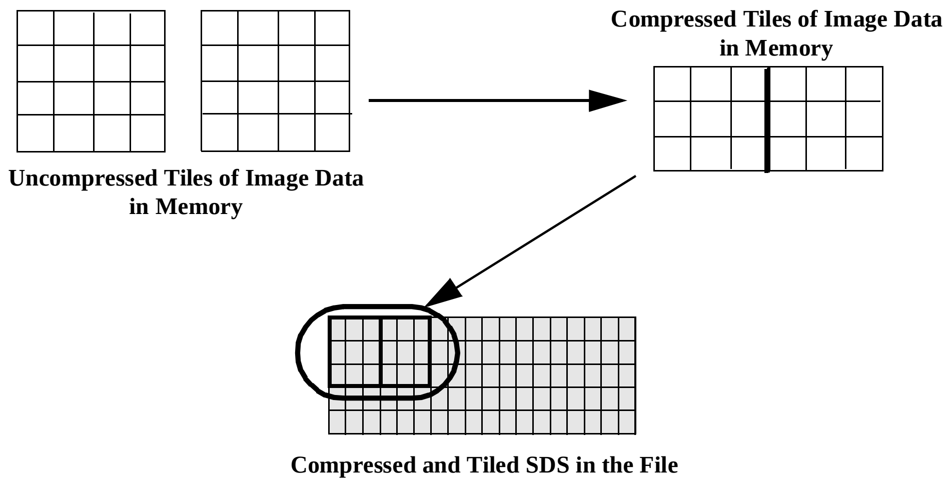
The efficiency of subsetting from chunked SDSs is partly determined by the size of the chunk - the smaller the chunk size, the more seeks will be necessary. Chunking can substantially improve I/O performance when data is read along the slowest-varying dimension. It can substantially degrade performance when data is read along the fastest-varying dimension.

14.3.4 Chunking with Compression

Chunking can be particularly effective when used in conjunction with compression. It allows subsets to be read (or written) without having to uncompress (or compress) the entire array.

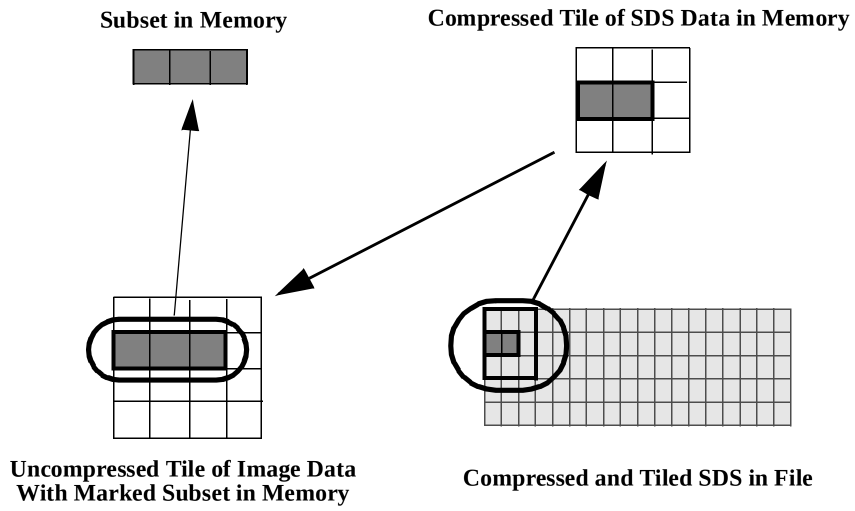
Consider the example of a tiled, two-dimensional SDS containing one million bytes of image data. Each tile of image data has been compressed as illustrated in the following figure.

FIGURE 14o **Compressing and Writing Chunks of Data to a Compressed and Tiled SDS**


Compressed Tiles of Image Data in Memory
Uncompressed Tiles of Image Data in Memory
Compressed and Tiled SDS in the File

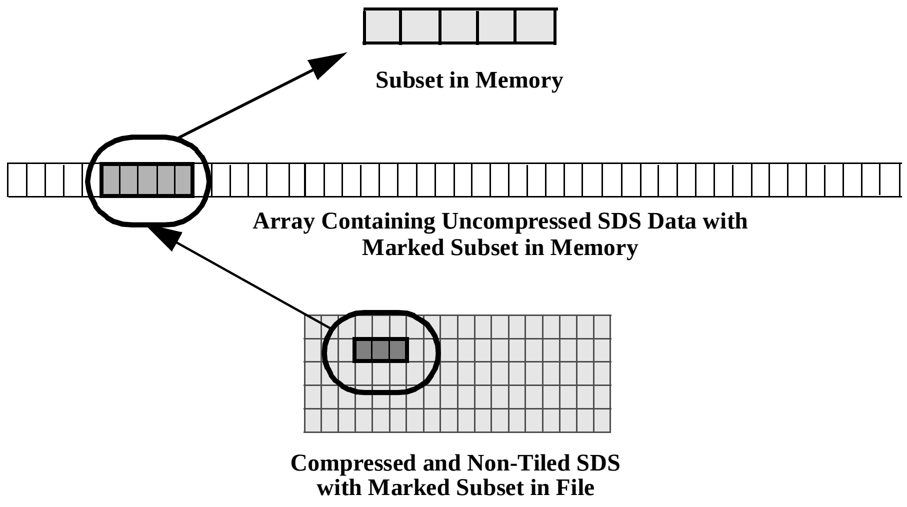
When it becomes necessary to read a subset of the image data, the application passes in the location of a tile, reads the entire tile into a buffer, and extracts the data-of-interest from that buffer.

FIGURE 14p **Extracting a Subset from a Compressed and Tiled SDS**


Subset in Memory
Compressed Tile of SDS Data in Memory
Uncompressed Tile of Image Data With Marked Subset in Memory
Compressed and Tiled SDS in File

In a compressed and *non-tiled* SDS, retrieving a subset of the compressed image data necessitates reading the entire contents of the SDS array into a memory buffer and uncompressing it in-core. (See Figure 14p.) The subset is then extracted from this buffer. (Keep in mind that, even though the illustrations show two-dimensional data tiles for clarity, this process can be extended to data chunks of any number of dimensions.)

FIGURE 14q **Extracting a Subset from a Compressed Non-Tiled SDS**


Subset in Memory
Array Containing Uncompressed SDS Data with Marked Subset in Memory
Compressed and Non-Tiled SDS with Marked Subset in File

As compressed image files can be as large as hundreds of megabytes in size, and a gigabyte or more uncompressed, it is clear that the I/O requirements of reading to and writing from non-tiled, compressed SDSs can be immense, if not prohibitive. Add to this the additional I/O burden inherent in situations where portions of several image files must be read at the same time for comparison, and the benefits of tiling become even more apparent.

NOTE: It's recommended that the **SDwritechunk** routine be used to write to a compressed and chunked SDS. **SDwritechunk** can perform this operation more efficiently than the combination of **SDsetcompress** and **SDwritedata**. This is because the chunk information provided by the user to the **SDwritechunk** routine must be retrieved from the file by **SDwritedata**, and therefore involves more computational overhead.

14.3.5 Effect of Chunk Size on Performance

The main concern in modelling data for chunking is that the chunk size be approximately equal to the average expected size of the data block needed by the application.

If the chunk size is substantially larger than this, increased I/O overhead will be involved in reading the chunk and increased performance overhead will be involved in the decompression of the data if it is compressed. If the chunk size is substantially smaller than this, increased performance and memory/disk storage overhead will be involved in the HDF library's operations of accessing and keeping track of more chunks, as well as the danger of exceeding the maximum number of chunks per file. (64K)

It is recommended that the chunk size be at least 8K bytes.

14.3.6 How Insufficient Chunk Cache Space can Impair Chunking Performance

The HDF library provides for the caching of chunks. This can substantially improve I/O performance when a particular chunk must be accessed more than once.

There is a potential performance problem when subsets are read from chunked datasets and insufficient chunk cache space has been allocated. The cause of this problem is the fact that two separate levels of the library are working to read the subset into memory and these two levels have a different "perspective" on how the data in the dataset is organized.

Specifically, higher-level routines like **SDreaddata** access the data in a strictly row-wise fashion - not according to the chunked layout. However, the lower-level code that directly performs the read operation accesses the data according to the chunked layout.

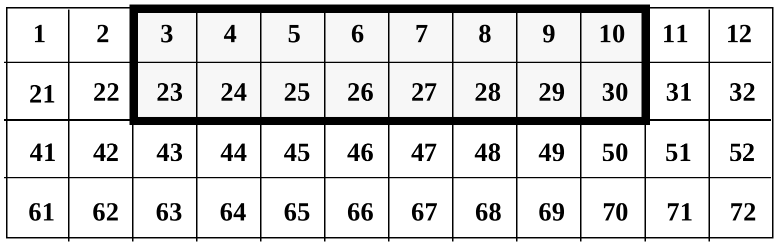
As an illustration of this, consider the 4 *x* 12 dataset depicted in the following figure.

FIGURE 14r **Example 4 *x* 12 Element Scientific Data Set**

1	2	3	4	5	6	7	8	9	10	11	12
21	22	23	24	25	26	27	28	29	30	31	32
41	42	43	44	45	46	47	48	49	50	51	52
61	62	63	64	65	66	67	68	69	70	71	72

Suppose this dataset is untiled, and the subset shown in the following figure must be read.

FIGURE 14s **2 *x* 8 Element Subset of the 4 *x* 12 Scientific Data Set**



1	2	3	4	5	6	7	8	9	10	11	12
21	22	23	24	25	26	27	28	29	30	31	32
41	42	43	44	45	46	47	48	49	50	51	52
61	62	63	64	65	66	67	68	69	70	71	72

As this dataset is untiled, the numbers are stored in linear order. **SDreaddata** finds the longest contiguous stream of numbers, and requests the lower level of the library code to read it into memory. First, the first row of numbers will be read:

```
3  4  5  6  7  8  9 10
```

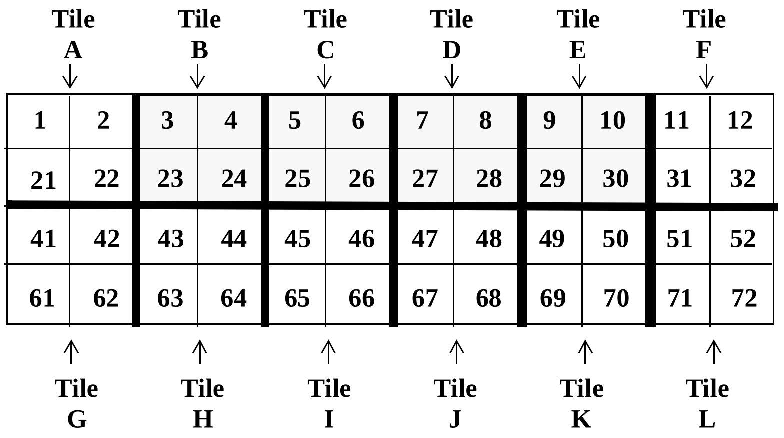
Then the second row:

```
23 24 25 26 27 28 29 30
```

This involves two reads, two disk accesses and sixteen numbers.

Now suppose that this dataset is tiled with 2 *x* 2 element tiles. On the disk, the data in this dataset is stored as twelve separate tiles, which for the purposes of this example will be labelled A through L.

FIGURE 14t **4 *x* 12 Element Data Set with 2 *x* 2 Element Tiles**



Also, the chunk cache size is set to 2.

A request is made to read the aforementioned subset of numbers into memory. As before, **SDreaddata** will determine the order the numbers will be read in. The routine has no information about the tiled layout. The higher-level code will again request the numbers "3, 4, 5, 6, 7, 8, 9, 10" from the lower level code on the first read operation.

In order to access those numbers the lower levels must read in four tiles: B, C, D, E. It reads in tiles B and C, retrieving the numbers "3, 4, 5, 6". However, as the cache space is now completely filled it must overwrite tile B in the cache to access the numbers "7, 8", which are in tile D. It then has to overwrite tile C to access "9, 10," which are in tile E. Note that, in each case, it does not use half of the numbers from the tile that it reads in, even though those numbers will be needed later.

Next, the higher-level code requests the second row of the subset - "23, 24, 25, 26, 27, 28, 29, 30." The lower-level code must *reread* tile B to access "23, 24", but tile B is no longer in the chunk cache. In order to access tile B, the lower-level code must overwrite tile D, and so on. By the time the subset read operation is complete, it has had to read in each of the tiles twice. Also, it has had to perform 8 disk accesses and has read 32 numbers.

Now consider a more practical example with the following parameters:

- A scientific data set has 3,000 rows and 8,400 columns.
- The target subset is 300 rows by 1,000 columns, and contains 300,000 numbers.

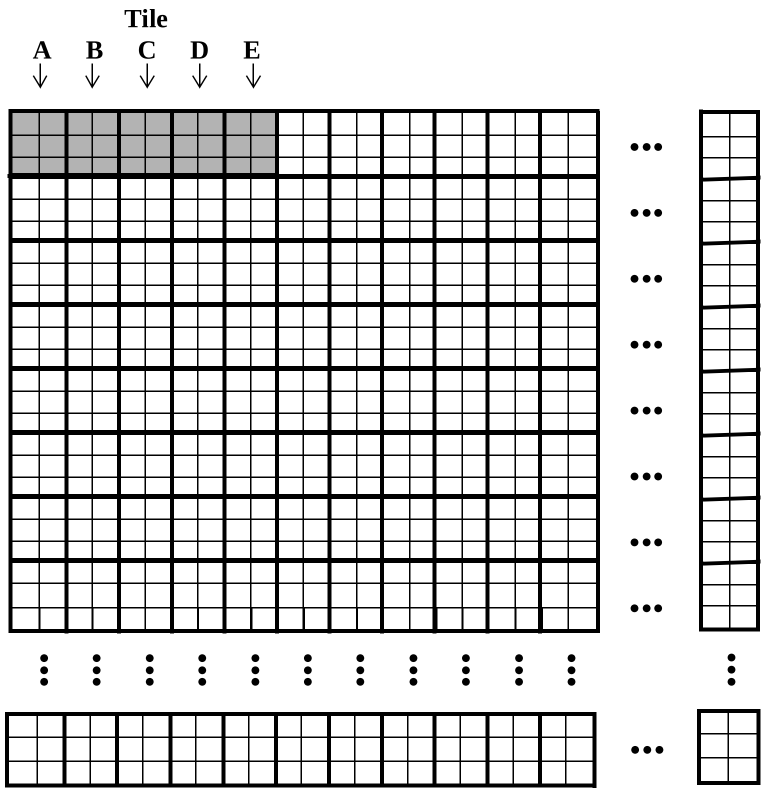
If the dataset is untiled the numbers are read into memory row-by-row. This involves 300 disk accesses for 300 rows, with each disk access reading in 1,000 numbers. The total number of numbers that will be read is 300,000.

Suppose the dataset is tiled as follows:

- The tile size is 300 rows by 200 columns, or 60,000 numbers.
- The size of the chunk cache is 2.

Each square in the following figure represents one 100 *x* 100 element region of the dataset. Five tiles span the 300 *x* 1,000 target subset. For the purposes of this example, they will be labelled A, B, C, D and E.

FIGURE 14u **5 200 *x* 300 Element Tiles Labelled A, B, C, D and E**



First, the higher-level code instructs the lower-level code to read in the first row of subset numbers. The lower-level code must read all five tiles (A through E) into memory, as they all contain numbers in the first row. Tiles A and B are read into the cache without problem, then the following set of cache overwrites occurs.

1. Tile A is overwritten when tile C is read.
2. Tile B is overwritten when tile D is read.
3. Tile C is overwritten when tile E is read.

When the first row has been read, the cache contains tiles D and E.

The second row is then read. The higher-level code first requests tile A, however the cache is full, so it must overwrite tile D to read tile A. Then the following set of cache overwrites occur.

1. Tile E is overwritten when tile B is read.
2. Tile A is overwritten when tile C is read.
3. Tile B is overwritten when tile D is read.
4. Tile C is overwritten when tile E is read.

For each row, five tiles must be read in. No actual caching results from this overwriting. When the subset read operation is complete, 300 * 5 = 1,500 tiles have been read, or 60,000 * 1,500 = 90,000,000 numbers.

Essentially, five times more disk accesses are being performed and 900 times more data is being read than with the untiled 3,000 *x* 8,400 dataset. The severity of the performance degradation increases in a non-linear fashion as the size of the dataset increases.

From this example it should be apparent that, to prevent this kind of chunk cache "thrashing" from occurring, the size of the chunk cache should be made equal to, or greater than, the number of chunks along the fastest-varying dimension of the dataset. In this case, the chunk cache size should be set to 4.

When a chunked SDS is opened for reading or writing, the default cache size is set to the number of chunks along the fastest-varying dimension of the SDS. This will prevent cache thrashing from occurring in situations where the user doesn’t set the size of the the chunk cache. Caution should be exercised by the user when altering this default chunk cache size.