Technical Report

Factors Affecting I/O Performance when Accessing Large Arrays in HDF5 on NCSA's TeraGrid Cluster[1]

Mike Folk and Vailin Choi

September 2005

Abstract

Achieving good I/O performance depends on a number of interacting components and how these are configured. These include the access patterns of the application itself, the architecture of the computing system, middleware such as I/O libraries, the type of file system, the mode of access, data set size, data storage layout, and the external storage configuration. This study of I/O performance on NCSA's TeraGrid computing system examines the role of access patterns and types of I/O with arrays stored in HDF5 using different storage layouts. Access patterns involves accessing data from an image by one or more column per access. Three different I/O modes are compared (serial, independent parallel, collective parallel), involving a relatively small and large arrays (48 MB vs.1 GB), and using both chunked and contiguous storage layouts. In the parallel modes, the effect of the number of nodes was also included. For the configuration used in the study both collective and independent I/O was better than serial I/O for 8 or more processors when accessing the large image, but not when accessing the small images. Collective I/O performed best when data was stored contiguously. Chunking improved all I/O modes for the access patterns tested, especially for serial I/O, and especially for the large image. Chunking was particularly effective for independent I/O when the contents of chunks corresponded closely to the access patterns.

1. Goals and focus of report

This paper has two goals: to identify some of the important factors that affect I/O performance in accessing large arrays on parallel systems, and to report on a study of some these factors on NCSA's TeraGrid system.

The TeraGrid study is part of an effort supported by the National Archives and Records Administration (NARA) Electronic Records Archive (ERA) program to understand issues of efficiency of I/O, accessibility and storage, as applied to 2D and 3D raster data

[1] This report is based upon work supported by the National Archives and Records Administration (NARA), funded through the National Science Foundation Cooperative Agreement PACI 96-190-19. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NARA.

and vector data, taking advantage of scientific data formats, and in particular taking advantage of alternate storage formats that the HDF5 scientific data format provides.

The study reported here focuses on performance in accessing 2D raster data serially and in parallel, and the effect of chunking on I/O performance in both modes. Experiments were done on two different parallel high end architectures, a Linux cluster (NCSA TeraGrid system) and a Symmetric MultiProcessor (SMP) system, the IBM P690 at NCSA. This report describes results from experiments with the NCSA TeraGrid system. A second report will describe results from the same experiments with NCSA IBM P690.

2. Factors affecting I/O performance on parallel systems

In earlier work [Jan 2004] we examined the I/O and storage benefits and costs of archiving data in a general purpose scientific format, where access was serial. In this study, we examine issues involved in parallel access to and from large images in HDF5. Images are stored in HDF5 as arrays, so this study focuses on array access.

Any computing system is built of many layers. Figure 1 illustrates some of the layers that can occur in a parallel system that can affect performance.

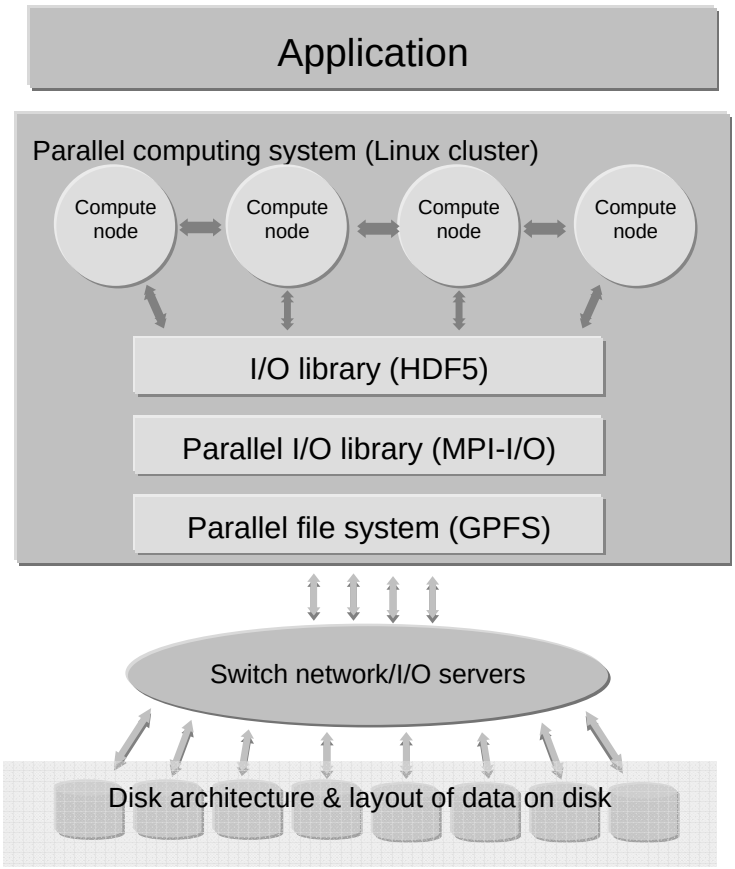

Figure 1. Layers affecting I/O performance.

For a given application, some of these layers are optional, but each layer is designed to help applications do something well, and some are particularly designed to facilitate I/O.

2.1 Parallel computing systems – SMP and Linux Cluster

In performing I/O, important differences among parallel systems include the number of processors and nodes used, how the nodes intercommunicate among themselves and with memory, and the type of file system. Parallel access is investigated on two different architectures with parallel file systems, NCSA's TeraGrid cluster, and NCSA's IBM P690 SP:

> **NCSA's TeraGrid cluster** ("mercury") is a Linux cluster with 887 IBM cluster nodes, each node consisting of 2 Itanium® 2 processors. Most of the nodes are equipped with four gigabytes (GBs) of memory per node, making them ideal for running memory-intensive applications. Processors within a node share memory, but no memory is shared across nodes, so I/O involving more than two processors must distribute data across nodes. The cluster is running SuSE Linux and is using Myricom's Myrinet cluster interconnect network. Of 130 TB of secondary storage, 90 TB are accessed through a Storage Area Network (SAN) and IBM's GPFS (General Parallel File System), 39 through a Network Shared Disk (NSD), and 1 TB through a Network File System (NFS).
>
> **NCSA's IBM P690 SP** ("copper") is a symmetric multiprocessor machine using IBM Power4 processors. There is one interactive node (16 processors, 32 GB memory), and 11 batch nodes. Each batch node consists of 32 processors, although only 16 processors were available on a given node for these experiments. 7 batch nodes have 64 GB memory, and 4 have 256 GB memory. All processors in a node share the same memory. I/O can be done from any number of processors within a node. Hence, as many as 16 processors can do I/O to and from the same memory. The systems runs the AIX operating system, and Gigabit Ethernet connects the systems. 35 TB of secondary storage are accessed using GPFS across a Storage Area Network.

2.2 Parallel file systems and I/O libraries – GPFS and MPI-IO

I/O on parallel systems often means many processors writing or reading data simultaneously, often to or from the same file. Traditional file systems were not designed to handle I/O in parallel, and hence I/O bottlenecks occur, especially when the I/O involves large datasets.

The most common approach to addressing this kind of bottleneck to implement parallel file systems – file systems that support simultaneous accesses to a file by separate processes. Large commercial systems often include their own parallel file systems, such as GPFS for IBM systems. In addition, in recent years some general purpose parallel file systems have emerged that are designed for a variety of platforms. These include the open source PVFS and Lustre parallel file systems.

This study focuses on IBM's GPFS file system because both of the systems in this study use GPFS. Because the architectures of the two systems are very different, the implementations of GPFS on those systems are also different. As a result, a particular

way of doing I/O may perform quite well on one system and poorly on the other.

I/O to GPFS can be either serial or parallel.

Serial I/O occurs when all access is done through one processor.

Parallel I/O occurs when I/O is done simultaneously by many processors.

Parallel I/O may be independent or collective.

Independent parallel I/O occurs when individual processors independently perform I/O, with no underlying coordination the I/O operations of the other processors. If the data to be accessed is separated on disk on a per-processor basis – that is, if there is no overlap of the data being accessed by different processors – then this independent parallel I/O generally an efficient way to perform parallel I/O.

Collective parallel I/O occurs when I/O is synchronized across many processors. For instance, when different processors request the same data, or they request data that is interspersed, independent I/O can result in reading the same data many times. In such cases, it may be possible for one process to do I/O on behalf of many processors, then redistribute the data to the processors that need it. This is called collective parallel I/O, and can yield significant improvements in I/O performance in appropriate circumstances.

Most systems, including those in this study, support MPI (Message Passing Interface), a library for message-passing across nodes and processor. The I/O component of MPI (MPI-IO) supports both independent and collective I/O.

2.3 I/O libraries and disk layout – HDF5 and chunking

Large objects, such as large images, are typically stored as large arrays on disk. Although it is common to store large arrays as a large contiguous stream of bytes on disk, it is possible to organize them in other ways as well, ways that could result in improved performance under some circumstances. HDF5 dataset storage illustrates some of these options.

An HDF5 dataset is a data array, together with metadata. A data array in HDF5 is a rectangular collection of values with a specified number of rows and columns. In this study we are concerned with the data array itself, not the metadata, which is relatively small compared with the size of the array. An HDF5 dataset can be any size and any number of dimensions, but in this study, the focus is on 2-dimensional datasets, which are commonly used for storing images.

HDF5 offers a number of options for organizing data within the file, to take advantage of the access patterns employed in reading and writing data. Among the HDF5 options are the way in which a HDF5 dataset is laid out on disk. A dataset can be stored as a linear array of contiguous bytes, or broken into separate chunks. It may be compressed, or even stored in a separate file. Many combinations of these storage options are also possible. This flexibility can lead to improvements in I/O performance, especially when the layout is chosen to match the type of file system involved in I/O, as well as the types of access patterns used to write and read subsets of data.

Chunking in HDF5. This study focuses on the use of chunking for array storage. HDF5 chunks are rectangular subsets of arrays. Chunks can be any size, but all chunks must be

the same size for an entire array. Figure 2 illustrates the use of chunking in HDF5.

Figure 2(a) illustrates an array with 12 rows, and some number of columns. In Figure 2(a), no chunking is used. The array is stored as a contiguous linear stream of values on disk, a row at a time, starting with the value in the upper left corner and ending with the value in the lower right corner.

In Figure 2(b) the array is chunked by row. Each chunk includes three entire rows. Within each chunk the rows are laid out on disk as a contiguous stream of values, a row at a time. The individual chunks may be located at different places on disk.

In Figure 2(c), the array is chunked by column. Within each chunk a number of entire columns are laid out on disk as a contiguous stream of values. Each row of a given chunk is only part of an actual row in the larger array. For instance, each row might consist of 3 values, corresponding to 3 columns. The chunks in Figure 2(d) consist of partial rows and partial columns.

In all four examples, the data is linearized in so-called *row-major* order. That is, the first row of elements in the dataset or chunk is stored contiguously, followed by the second row, the third row, and so forth. Elements in a given column, then are separated in storage by whatever the length of a row is. This storage organization can have important implications for access performance. Access by row should, in theory, be much faster than access by column.

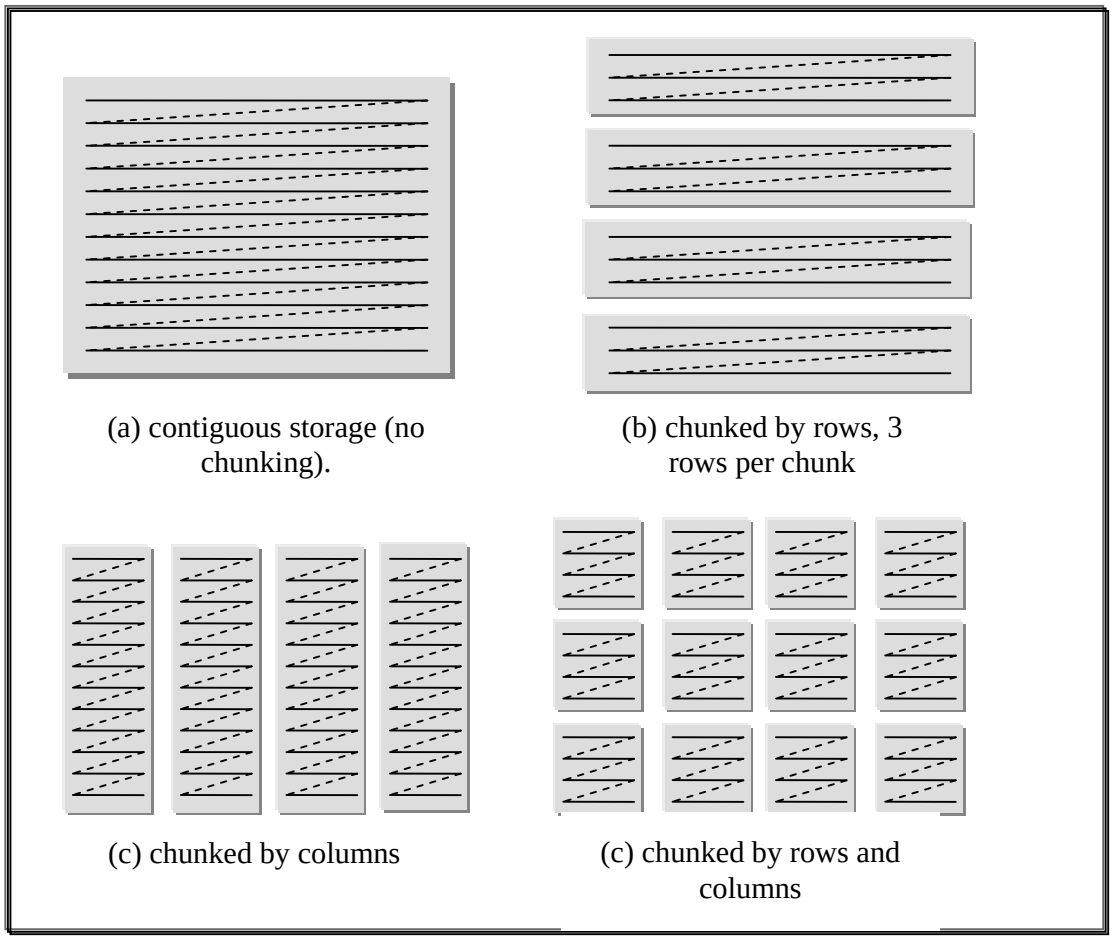

Figure 2. Chunking options in HDF5.

Size and shape of array access. A rule of thumb for accessing any secondary storage is the "bigger is better almost all of the time." That is, it is better to get as much as is needed in one access than to do many small accesses from the same dataset.

A related factor that affects I/O performance is the size of buffers used to transfer data – that is, the buffers in memory that are used to stage data when it is moved between file and memory. When transfer buffers are larger, a greater amount of data can be loaded in one access. If a desired selection involves a sufficiently large portion of the data that is loaded into the buffer, performance can be favorably affected.

In the case of chunking, accesses that are small (relative to chunk size) can be particularly bad for performance because chunks must always be loaded in their entirety from disk. If the size of a selection is substantially less than the size of a large chunk, much more data will be loaded than is actually needed.

For instance, in the experiments performed in this study a chunk size of 525 columns is used. If a selection includes only 1 column, then 524 columns are loaded unnecessarily. However, if ultimately all of the columns in a chunk are actually accessed, even if they are accessed one at a time, the performance penalty can be mitigated by the use of chunk caching, which involves keeping chunks in memory in anticipation of later use.

In general, a good rule of thumb for chunking is that the size of a chunk should match reasonably closely with the amount of data to be used from the chunk for any given reading or writing of the chunk. If chunk caching is used, such accesses need not be simultaneous, as long as they happen while the chunk remains in the cache.

2.4 Interacting factors, for better or worse

We have seen that a number of capabilities are available to contribute to good computing and I/O performance. In many cases, these features can be combined to achieve effective performance. We have learned, however, that the interactions among these layers can negatively affect I/O performance. This study is designed to help us identify both the ways in which certain features can improve performance, and also to identify cases where performance is not improved or even becomes worse. Table 1 lists the types of interactions we are studying and the factors under consideration.

Interactions	**Factors**
Application and computing system	Shared vs. distributed memory Number of nodes and processors used by application
File system and I/O libraries	Type and implementation of file system Serial vs. parallel I/O Independent parallel vs. collective parallel I/O (e.g. MPI-IO)
I/O library and layout of data on disk	Chunked vs., contiguous layout of arrays Sizes of chunks
Application I/O, I/O library and data layout	Array size Shape of array access (rows vs. columns) Size of access (number of rows, columns)

Table 1. I/O performance in accessing large arrays on parallel systems is influenced at a number of factors, including computer system architecture, file system and type of io, size and layout of data on disk, and size and shape of accesses.

3. Experiments

In this study, we look at the effect of different format layouts on performance in parallel environments. The focus is in reading subsets of data from large, monolithic datasets. When large datasets are accessed, it is common to read or write from only part of the dataset, especially in parallel computing environments. Parallel processing often involves subdividing a dataset and parceling out its pieces to different processors.

The experiments are designed to evaluate several factors:

1. The value of chunking in improving performance when accessing “against the grain” of normal contiguous storage
2. The effectiveness of accessing large selections over small selections
3. The relative effectiveness of independent vs. collective parallel vs. serial access

4. The effect of image size on I/O performance, in the context of the other variables.

3.1 Methodology

These experiments all involved operations that read and write entire datasets, but in pieces –by accessing a number of columns at a time. Essentially no computation is done, so the timings primarily measured read and write time on selections of rows and columns.

Two sizes of arrays were used in the investigation:

Dataset	Dimensions	Size (bytes)
"ORIGDOQ" – Original DOQ from USGS[2]	7,639 rows x 6,308 cols	48,203,676 (48 MB)
"BIGDOQ" – Large DOQ made by stitching together 21 DOQs in one large column	160,419 rows x 6,308 cols	1,011,939,916 (1 GB)

Both images were converted from their native format to HDF5.

To measure the effects of chunking, the images were stored as HDF5 datasets using contiguous storage (no chunks) and chunking by column. These storage organizations are illustrated in Figure 2(a) and (c), respectively.

3.1.1. Serial vs. collective parallel vs. independent parallel I/O

Three kinds of I/O performance were measured for each kind of chunking – serial I/O, collective parallel I/O and independent parallel I/O. In the case of serial I/O one processor was used to read and write all data in a dataset. In the parallel cases, varying numbers of processor (from 1 to 12) were used to read and write all the data, where each processor read different selections from the dataset.

3.1.2. Chunking vs. contiguous storage

Column access was measured, with and without chunking. Since contiguous (non-chunked) images are stored in row major order, and the row length is very large, one would expect access by column to be relatively slow for non-chunked datasets.

To optimize access, the chunked datasets were organized in chunks that span entire columns. Chunks were 525 columns wide and consisted of all rows in the respective dataset: 7,639 rows for the small DOQ and 160,419 rows in the large DOQ.

To summarize, four HDF5 datasets were used, with the characteristics shown in

[2] Sample DOQ from USGS at http://edc.usgs.gov/geodata/samples.html.

Dataset	Dimensions/chunks
ORIGDOQ_contig.h5	7,639 x 6,308 contiguous array
ORIGDOQ_chunk.h5	13 7639x525 chunks
BIGDOQ_contig.h5	160,419 x 6308 contiguous array
BIGDOQ_chunk.h5	13 160,419x525 chunks

Table 2.

Dataset	Dimensions/chunks
ORIGDOQ_contig.h5	7,639 x 6,308 contiguous array
ORIGDOQ_chunk.h5	13 7639x525 chunks
BIGDOQ_contig.h5	160,419 x 6308 contiguous array
BIGDOQ_chunk.h5	13 160,419x525 chunks

Table 2. Characteristics of datasets. Two datasets were used, one with a relatively small (7,639 x 6,308) image, one with a large (160,419 x 6,308) image. Each image was stored in two ways, as a contiguous array and using chunks consisting of 525 columns.

3.1.3. Large selections vs. small selections

Columns were access in selection sizes of 1 column and 525 columns, for both contiguous and chunked datasets, for both BIGDOQ and ORIGDOQ. In HDF5, the use of different selection sizes results in transfer buffer sizes ranging from very small to very large, as shown in Table 3. Because a larger transfer buffer allows a greater amount of data to be loaded in one access, performance should improve.

Table 3. Selection sizes used in the tests, together with corresponding transfer buffer size for each selection.

Dataset	Selection size	Transfer buffer size
ORIGDOQ_contig.h5	1 column	7k
ORIGDOQ_contig.h5	525 columns	4M
BIGDOQ_contig.h5	1 column	160K
BIGDOQ_contig.h5	525 columns	84M

3.2 Tests

In all tests the entire dataset was always read and written, using a series of reads/writes.

For a given test, each read/write always involved the same number of columns. Experiments were coded in C and parallel I/O was done using the MPI programming model. Each experiment was run six times; in the results shown below the maximum performance for the six runs is reported.

Table 4 describes the variables used in these tests. As the table shows, selections of 525 columns were tested for contiguous and chunked storage, and for both the large and small images. Selections of 1 column were performed for both large and small chunked datasets.

Storage scheme:	**Contiguous**				**Chunked**			
Image size:	**ORIGDOQ**		**BIGDOQ**		**ORIGDOQ**		**BIGDOQ**	
No. columns selected:	**1**	**525**	**1**	**525**	**1**	**525**	**1**	**525**
Serial		●		●	●	●	●	●
Independent parallel		●		●	●	●	●	●
Collective parallel		●		●	●	●	●	●

Table 4. Experiments performed doing column access and varying storage scheme, image size, selection size, and type of serial and parallel access.

The tests differed according to storage scheme (contiguous vs. chunked) and type of parallelization (serial vs. independent-parallel vs. collective parallel):

1. Serial-column-contiguous
    a. One processor read/write based on 525-column selections.
2. Parallel-independent-column-contiguous
    a. Divide total number of 525-column selections among processors
    b. Processors read/write with independent calls
    c. Any remaining columns are read/written by last processor
3. Parallel-collective-column-contiguous
    a. Divide total number of 525-column selections among processors
    b. Processors perform collective calls
    c. Any remaining columns are read/written by processor 0
4. Serial-column-chunked
    a. One processor read/write based on selection of 1- and 525-column selections.
5. Parallel-independent-column-chunked
    a. Divide total number of 1- and 525-column selections among processors
    b. Processors read/write with independent calls
    c. Any remaining columns are read/written by last processor

6. Parallel-collective-column-chunked
    a. Divide total number of 1- and 525-column selections among processors
    b. Processors perform collective calls
    c. Any remaining columns are read/written by processor 0

3.3 Results

The results are presented in three sections: results of experiments with the contiguously stored images, results with chunked images, and a comparison of contiguous vs. chunked results.

3.3.1. Contiguous storage results

When contiguous storage is used, how do the three modes of access compare? Does image size make a difference? Figure 3 shows results for the contiguous storage tests. In all cases, all selections were 525 columns. Detailed findings are described in Table 5 - Table 8. Notable findings are:

- Parallel I/O performance was better than serial only for the large image, and was best for larger numbers of I/O processors.
- Collective I/O was better than independent I/O only for 8 and 12 processors. Otherwise the difference was not appreciable.
- Serial performances for the small image was better than either parallel mode for 1, 2 and 4 processors.
- Serial performance was far better for the small image than for the large image.
- Image size did not conclusively affect performance for either type of parallel access. For both parallel access modes, the curves for the small and large images do not appear to be significantly different.

Comment [mf1]: Ask Vailin about..

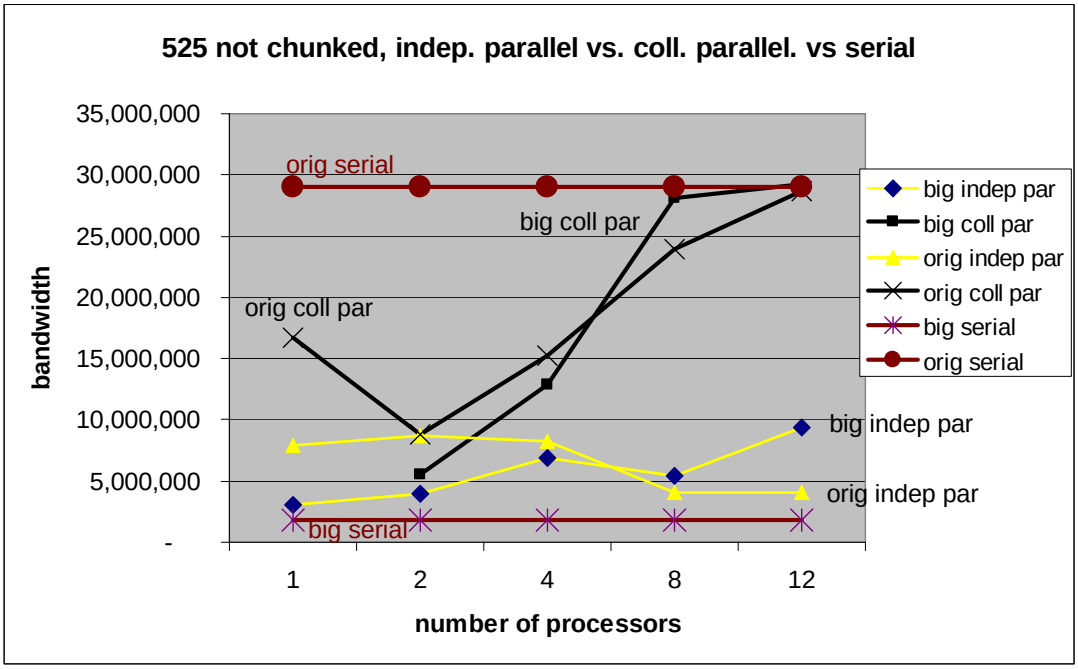

Figure 3. Contiguous storage, read/write results for ORIGDOQ (orig) and BIGDOQ (big), 525 columns per access, serial vs. collective parallel vs. independent parallel I/O. Bandwidth is in bytes/sec.

File	Results
BIG_contig	Parallel performed about the same as serial for 1 processor and improved to about 5 times faster for 12 processors.
ORIG_contig	Parallel performance was considerably worse than serial, and grew worse as the number of processors increased.
Big vs. orig	Orig. serial performed 3-10 times faster than the other three. Big serial worse than all the others.

Table 5. Independent parallel vs. serial, contiguous storage for BIG and ORIG DOQ, 525 columns accessed per selection.

File	Results
BIGDOQ_contig	Parallel HDF performed better than serial in all cases and rose to about 15 times serial for 12 processors.[3]
ORIGDOQ_contig	Parallel performance was considerably worse than orig serial, but improved as the number of processors increased, and was about equal for 12 processors.

Table 6. Collective parallel vs. serial, contiguous storage for BIG and ORIG DOQ, 525 columns accessed per selection.

File	Results
Both files	For 1-2 processors, no appreciable difference is obvious, but for larger numbers of processors, collective performance improves substantially over independent, reaching 4-7 times faster for 8 and 12 processors.

Table 7. Collective parallel vs. independent parallel, contiguous storage for BIG and ORIG DOQ, 525 columns accessed per selection.

File	Results
Both	Serial performance on ORIG was 17 times better than on BIG.
Both	Neither type of parallel access was affected appreciably by the size of the file. Collective performance did improve as the number of processors increased.

Table 8. Effect of file size on type of access, contiguous storage for BIG and ORIG DOQ, 525 columns accessed per selection.

3.3.2. Chunked storage results

These tests examine the effects on performance of image size (BIGDOQ vs. ORIGDOQ), selection size (1 column vs. 525 columns) and I/O method (serial vs. collective parallel vs. independent parallel) in accessing datasets stored in chunks of size 7639x525 for the small ORIGDOQ, and chunks of size 160,419x525 for the large BIGDOQ.

Detailed findings are described in the following sections. Notable findings are:

a. When the selection was equal to the chunk size (525 columns):

[3] There is no record for parallel with one processor, as an error occurred in measuring this case.

- Serial access was about 35% faster for BIG than for ORIG.
- For 12 processors, parallel performance was 2-3 times better than serial on both BIG and ORIG.
- Both parallel I/O modes performed better for BIG than for ORIG, but independent I/O became less better as the number of processors increased. Collective I/O displayed the opposite pattern.

b. When the selection was 1 column:
   - For 1 processor there was little difference among the three access modes.
   - For 12 processors independent parallel was between 2.3 and 3.2 times better than either of the other two, for both image sizes.
   - Collective parallel peaked at 8 processors, then dropped 30% for 12 processors, for both image sizes.
   - Image size did make a difference – access to BIG was generally faster. Serial was 55% better. For 4 or more processors, independent parallel was 74-94% faster, and collective was 58-77% faster.

c. Comparing large (525 column) vs. small (1 column) accesses, results were quite mixed, except for the collective I/O on 12 processors. In this case, collective I/O performance was about twice as fast for large accesses than for small accesses.

a. Large selection, independent vs. collective vs. serial access

How do the three different modes of access perform on the two datasets when large (525 columns) accesses are performed? Figure 4 shows the bandwidths for each mode of access, and Figure 5 shows the ratios of independent/collective, independent/serial, and collective/serial performances for large and small datasets.

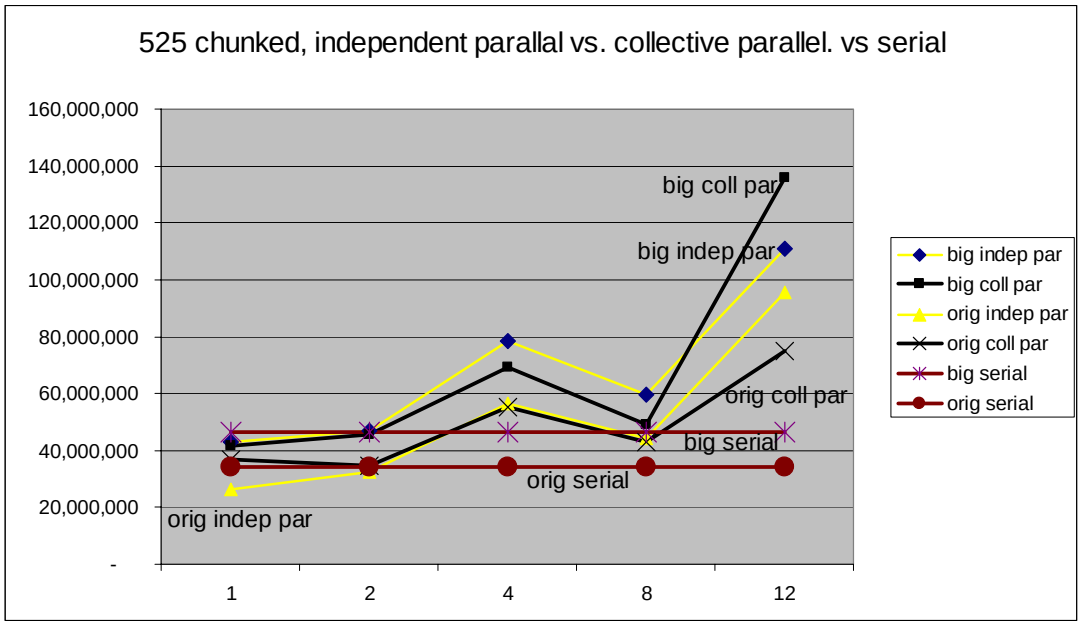

Figure 4. Chunked storage, read/write results for ORIGDOQ (orig) and BIGDOQ (big), 525 columns per access, serial vs. collective parallel vs. independent parallel I/O.

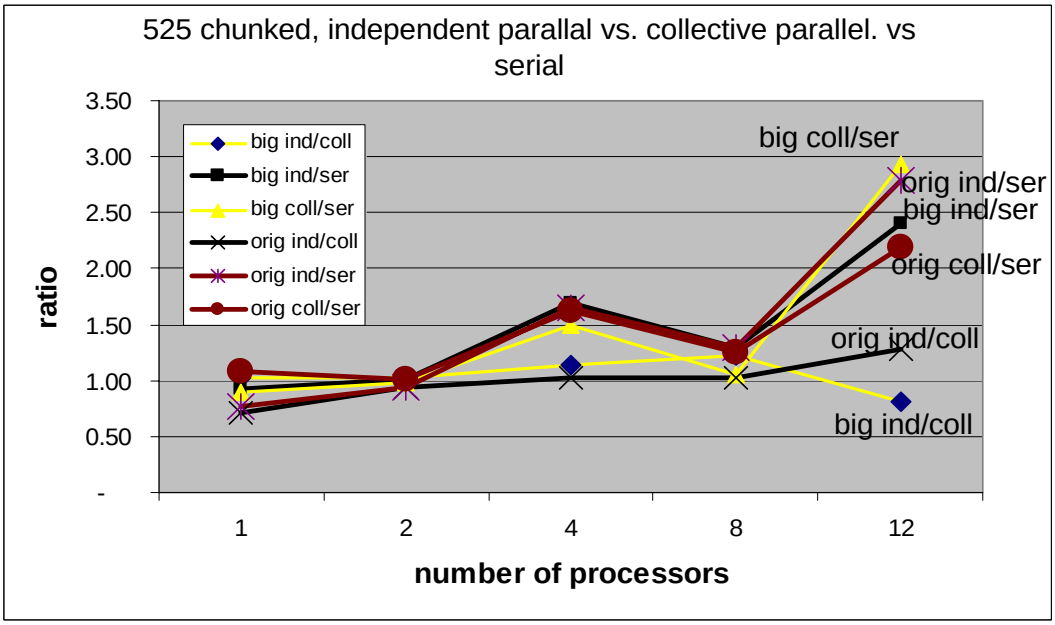

Figure 5. Chunked storage, read/write results for ORIGDOQ (orig) and BIGDOQ (big), 525 columns per access, showing the ratio of independent/collective, independent/serial, and collective/serial performances.

Cols/seln	File	Results
525	*BIG_chunk*	All modes performed about the same for 1 processor. Collective and independent performed better than serial for 12 processors, with collective parallel the best at about 3 times the performance of serial.
525	*ORIG_chunk*	All modes performed about the same for 1 processor. Collective and independent performed better than serial for 12 processors, with independent parallel the best at about 3 times the performance of serial.
525	*Big vs. orig chunk*	Serial access was about 35% faster for the BIG than for ORIG. Independent I/O 62% better for 1 processor when accessing BIG vs. ORIG, but steadily diminished to 16% better for 12 processor. Collective I/O was mixed. BIG was always better, and was 80% faster for 12 processors, but less than 15% better for 1 and 8 processors.

Table 9. Independent vs. collective vs. serial, chunked storage for BIG and ORIG DOQ, 525 columns accessed per selection. Based on Figure 4.

b. Small selection, independent vs. collective vs. serial access

How do the three different modes of access perform on the two datasets when small (1 column) accesses are performed? To compare the different modes, the ratio of bandwidth for independent/serial, independent/collective, and collective/serial are computed for both image sizes (Figure 6). Table 10 describes the results. To compare performance based on the two image sizes, the ratios of big/orig performance are computed for each mode
(

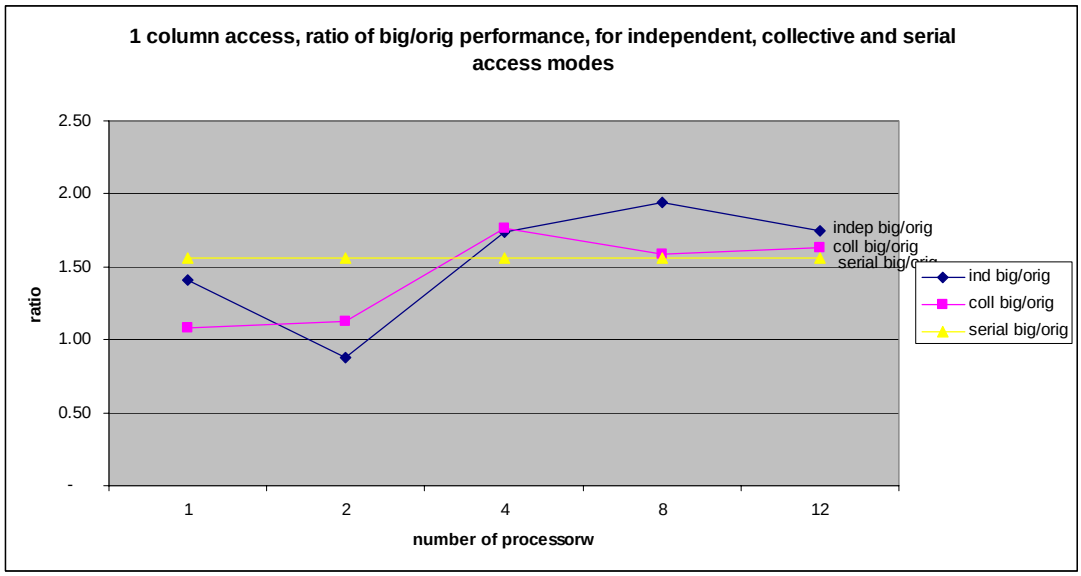

Figure 7). Table 11 describe the results.

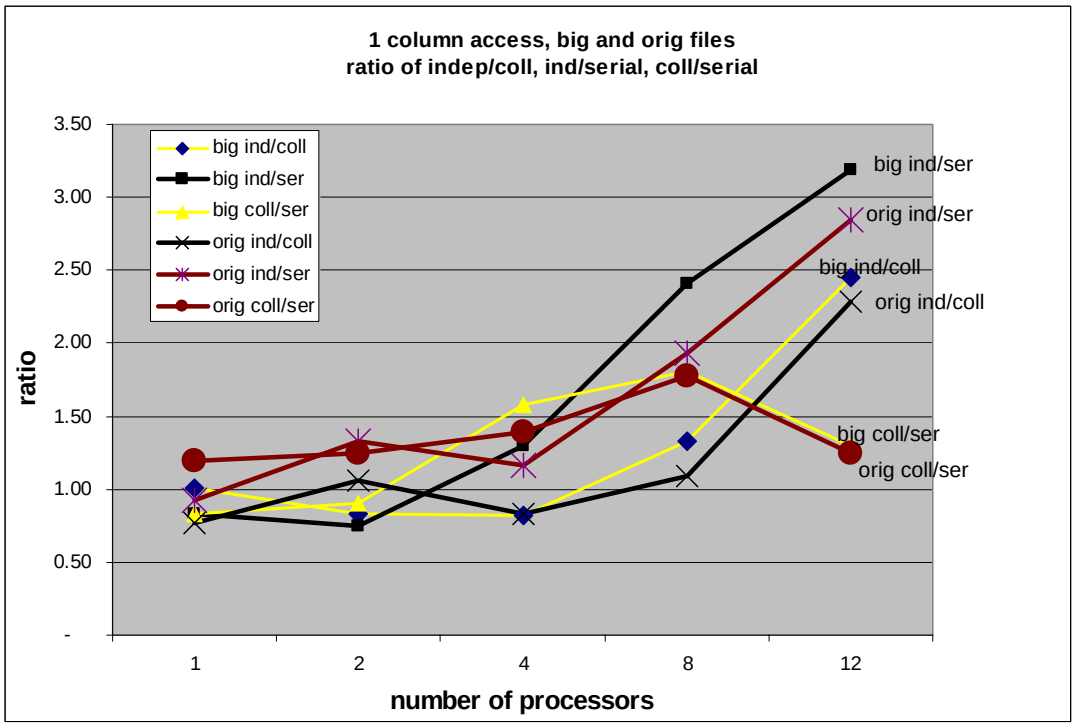

Figure 6. Chunked storage, 1 column read/write results for BIGDOQ and ORIGDOQ, showing the ratio of independent/collective, independent/serial, and collective/serial performances.

Cols/seln	File	Results
1	*BIG_chunk*	For 1 processor serial performed about 20% better than both parallel modes. For 12 processors, independent parallel was best at 3.2 times serial and 2.5 times collective. Independent parallel increased steadily with the number of processors, but collective parallel peaked at 8 processors, then dropped 30% for 12 processors.
1	*ORIG_chunk*	For 1 processor, collective was slightly better than serial, which was slightly better than independent parallel. For 12 processors independent parallel again best at 2.8 times serial and 2.3 times collective. Independent parallel increased from 4 to 8 to 12 processors, but collective peaked at 8 processors, then dropped 30% for 12 processors.

Table 10. Independent vs. collective vs. serial, chunked storage for BIG and ORIG DOQ, 1 column accessed per selection. Based on Error! Reference source not found.**.**

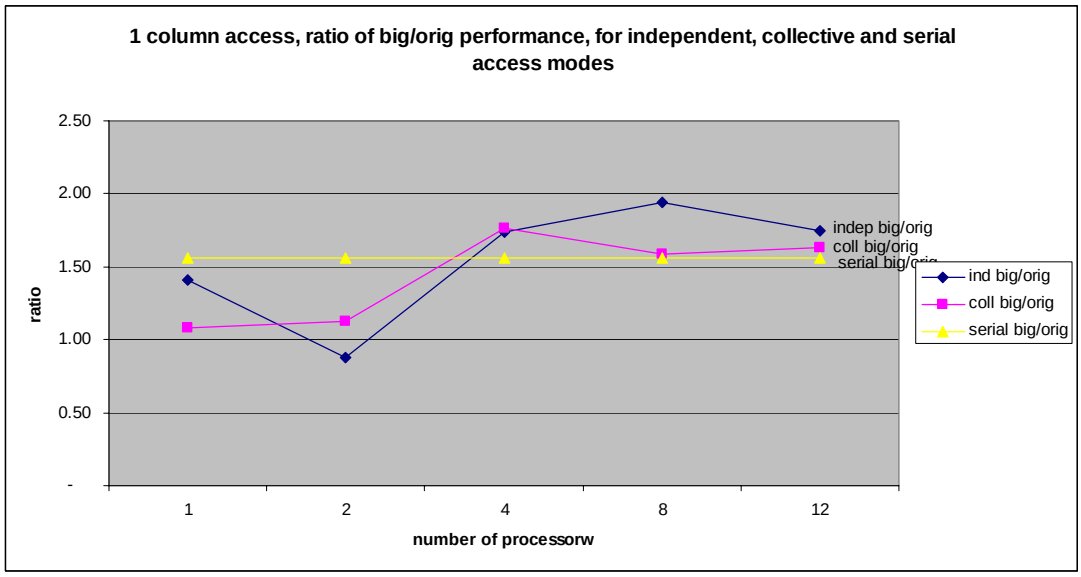

Figure 7. Chunked storage, 1 column read/write results for BIGDOQ and ORIGDOQ, showing the ratio of showing ration of big/orig access times for independent, collective, and serial access modes.

1	*BIG vs. ORIG*	Serial access was 55% better for BIG than for ORIG. Independent access was 74-94% better for BIG on 4 or more processors. For 1-2 processors, the results were

		mixed. Collective access was 58-77% better for BIG on 4 or more processors. For 1-2 processors, the results were slightly better for BIG.

Table 11. Comparison of performance on big image vs. original image when 1 column selection is performed.

c. Large vs. small accesses

How do the performances of large (525 columns) vs. small (1 column) access compare? To measure this, the ratio of I/O performance for large vs. small access were computed, for both image sizes (Figure 8 and Figure 9).

These results were quite mixed, except for the case of collective I/O for the maximum number of processors. In the 12-processor case, collective I/O performed better in both instances.

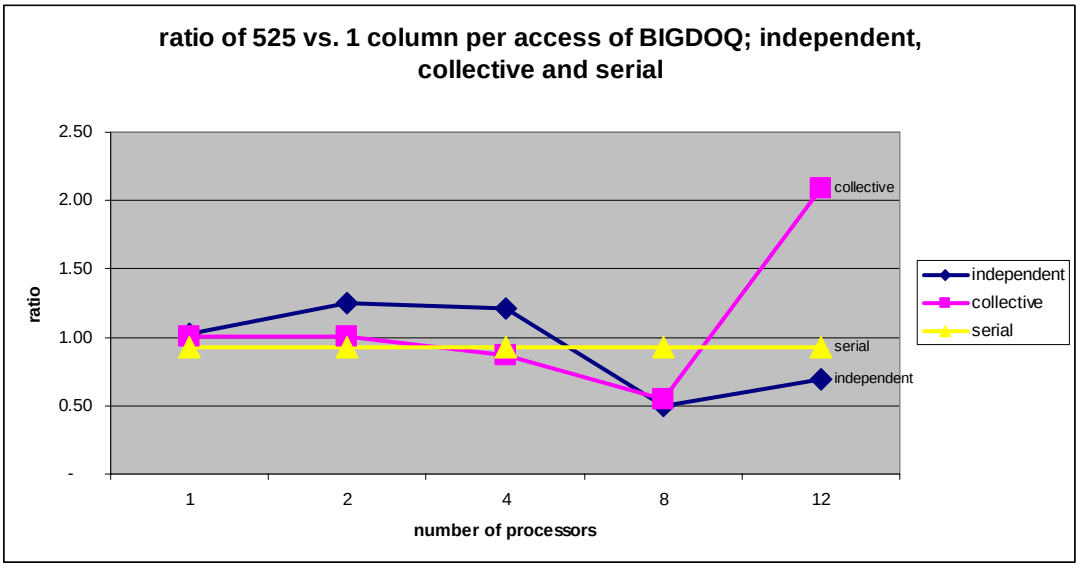

Figure 8. Chunked storage read/write results for BIGDOQ, showing the ratio of access times at 525-columns vs. 1 column per access, for independent, collective and serial access modes.

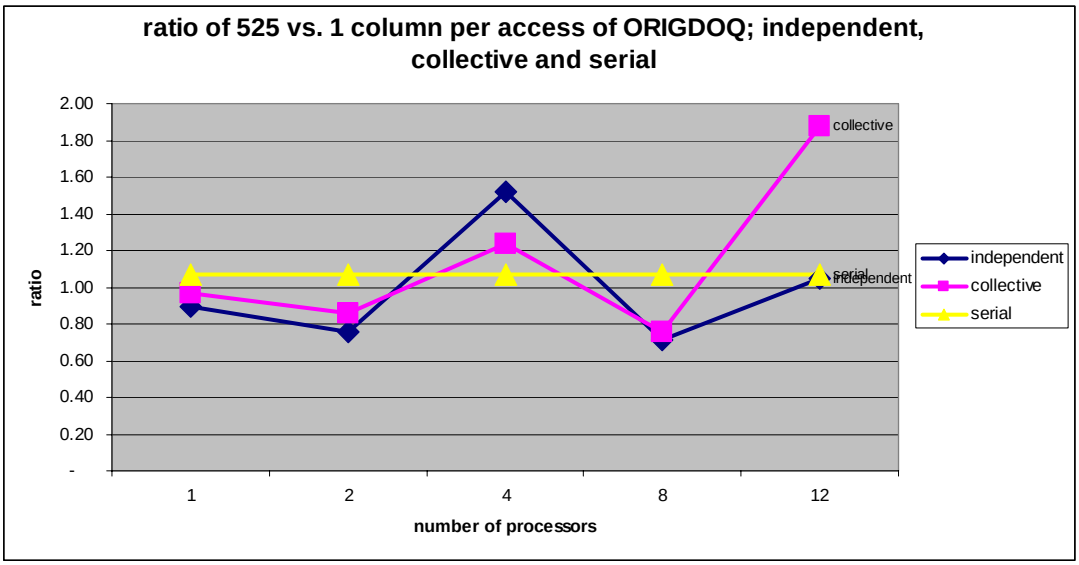

Figure 9. Chunked storage read/write results for ORIGDOQ, showing the ratio of access times at 525-columns vs. 1 column per access, for independent, collective and serial access modes.

3.3.3. Comparing chunked vs. contiguous results

What difference does chunking make in I/O performance? To measure the effect of chunking, the ratio of I/O performance for access time to chunked vs. contiguous data were computed, for both the BIGDOQ and ORIGDOQ (Figure 10 and Figure 11).

For BIGDOQ (Figure 10):

- In all cases, the ratios are greater than 1, indicating that chunking improved I/O performances in all of the cases tested.
- Serial I/O performance was more than 25 times better with chunking.
- Independent I/O performance was consistently between 10 and 15 times better with chunking.
- Collective performance also better, but less so. The difference between independent and collective results reflects the fact that collective I/O handles the scatter/gather operations required by contiguous storage considerably better than either independent or serial access modes.

For ORIGDOQ (Figure 11):

- Results suggest that collective and serial I/O are both less affected by chunking for small datasets.
- For serial I/O the improvement from chunking is tiny, reflecting the greater locality of data elements to be accessed in the smaller image.
- The other notable result is the steady and dramatic improvement of independent I/O. In this case the fact that each chunk is accessed completely by only one processor leads to the kind of parallelism that can give the best performance.

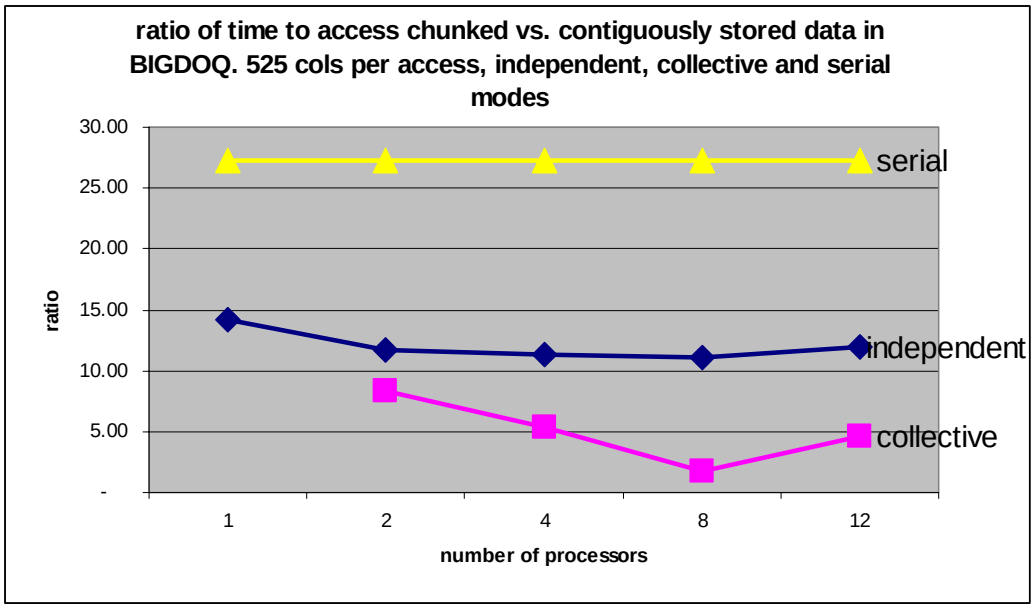

Figure 10. The effect of chunking on performance, for BIGDOQ. The graph shows the ratio of time to access chunked vs. contiguously stored data in BIGDOQ. 525 cols per access, independent, collective and serial modes.

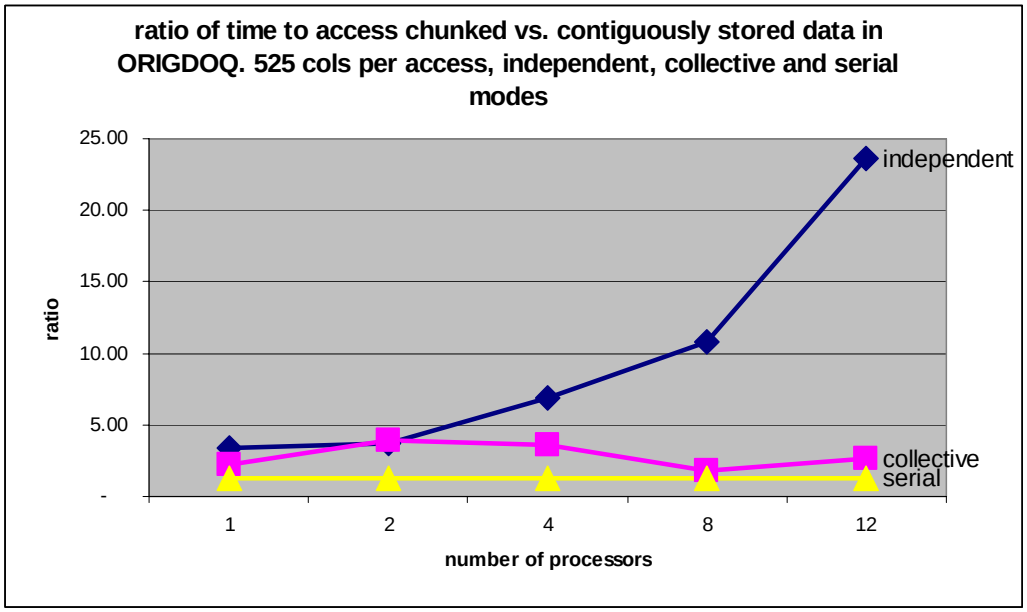

Figure 11. The effect of chunking on performance, for ORIGDOQ. The graph shows the ratio of time to access chunked vs. contiguously stored data in ORIGDOQ. 525 cols per access, independent, collective and serial modes.

4. Summary

Achieving good I/O performance depends on a number of interacting factors.

Both collective and independent parallel I/O performed better than serial in all cases where more than 8 processors were used and the image was large. For small images, however, serial I/O was often better than either parallel mode.

Collective I/O, which is designed to take advantage of situations where data requests across processors have collective locality, did indeed perform best among the three access modes in the contiguous case; that is when the data was spread out relative to any given processor's I/O request, but exhibited locality relative to the collective requests of all processors.

Chunking improved I/O in all cases tested. Chunking provides locality, particularly when the data being sought in a given access is in one or a few chunks – in this case, all in one chunk. Predictably, serial performance was greatly improved for the large image, mainly because of this locality. For the small image, where non-chunked data is much less spread out, the difference was far smaller.

In the chunking case, collective performance was not dramatically better than the other modes, in contrast to the contiguous case. This is because the advantages of collective I/O in coalescing requests from different processors are diminished by the locality enforced by chunking.

With chunking, independent I/O can be particularly effective when chunks contain exactly the data that is requested in a given access. This was demonstrated by the dramatic improvement of independent I/O on chunked data as the number of processors increased.