Evaluating I/O Characteristics and Methods for Storing Structured Scientific Data

Avery Ching[1], Alok Choudhary[1], Wei-keng Liao[1], Lee Ward[2], and Neil Pundit[2]

[1]Northwestern University
Department of EECS
Evanston, IL 60208-3118 USA
{aching, choudhar, wkliao}@ece.northwestern.edu

[2]Sandia National Laboratories
Scalable Computer Systems Department
Albuquerque, NM 87185-1110 USA
{lee, pundit}@sandia.gov

Abstract

Many large-scale scientific simulations generate large, structured multi-dimensional datasets. Data is stored at various intervals on high performance I/O storage systems for checkpointing, post-processing, and visualization. Data storage is very I/O intensive and can dominate the overall running time of an application, depending on the characteristics of the I/O access pattern. Our NCIO benchmark determines how I/O characteristics greatly affect performance (up to 2 orders of magnitude) and provides scientific application developers with guidelines for improvement. In this paper, we examine the impact of various I/O parameters and methods when using the MPI-IO interface to store structured scientific data in an optimized parallel file system.

1. Introduction

There is a class of scientific simulations that compute on large, structured multi-dimensional datasets which must be stored at numerous time steps. Data storage is necessary for visualization, snapshots, checkpointing, out-of-core computation, post processing [11], and numerous other reasons. Integrated Parallel Accurate Reservoir Simulation (IPARS) [12] is one such example. IPARS, a software system for large-scale oil reservoir simulation, computes on a three-dimensional data grid at every time step with 9,000 cells. Each cell in the grid is responsible for 17 variables. A total of 10,000 time steps will generate approximately 6.9 GBytes of data. Another example of scientific computing on large structured datasets is the Advanced Simulation and Computing (ASC) FLASH code. The ASC FLASH code [7], an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamics equations for studying nuclear flashes on neutron stars and white dwarfs, stores 24 variables per cell in a three-dimensional data grid. Storing time step data in structured data grids for applications like IPARS and ASC FLASH requires using an I/O access pattern, which contains both a memory description and a matching file description. When individual variables are computed and stored, the memory and file descriptions generated for the resulting I/O access pattern may have contiguous regions as small as a double (usually 8 bytes). Numerous studies have shown that the noncontiguous I/O access patterns evident in applications as IPARS and FLASH are common to most scientific applications [1, 6]. Most scientific applications use MPI-IO natively or through higher level I/O libraries such as pNetCDF [9] or HDF5 [8].

Cluster computing has been rapidly growing as the leading hardware platform for large-scale scientific simulation due to cost-effectiveness and scalability. While I/O has traditionally been a bottleneck in PCs, attempting to service the I/O requirements of an entire cluster has only augmented this problem to a much larger scale. Parallel file systems have helped to attain better I/O performance for clusters by striping data across multiple disks and are commonly used in most large-scale clusters [10, 5, 13, 2, 16].

In this paper, we generalize noncontiguous I/O access for storing scientific data in a modern parallel file system and evaluate the effects of varying three I/O characteristics: region count, region size and region spacing. We created the noncontiguous I/O benchmark, NCIO, to help application designers optimize their I/O algorithms. NCIO tests various I/O methods (POSIX I/O, list I/O, two phase I/O, and datatype

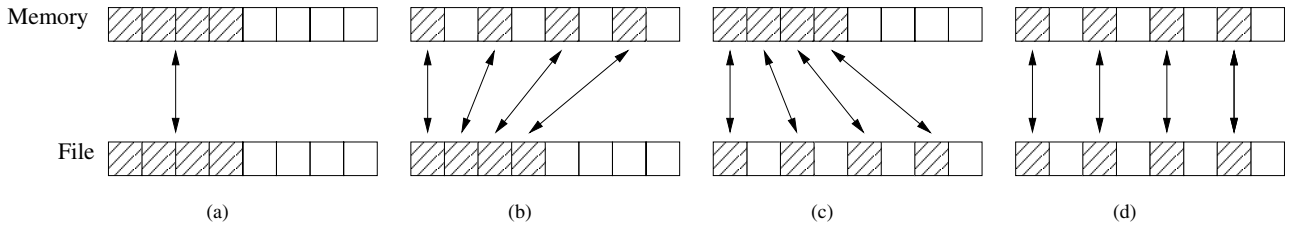

Figure 1. Various I/O access cases. (a) refers to contiguous in memory and file (c-c). (b) refers to noncontiguous in memory and contiguous in file (nc-c). (c) refers to contiguous in memory and noncontiguous in file. (d) refers to noncontiguous in memory and file (nc-nc).

I/O methods) in all I/O cases (c-c, nc-c, c-nc, nc-nc) with the important I/O characteristics of region count, region size, and region spacing. The results of our testing provides guidelines that can help scientific application developers understand how their I/O algorithms can significantly affect I/O performance between 1 to 3 orders of magnitude. We offer a set of guidelines for improving I/O performance based on our results.

Our paper is organized as follows. In Section 2, we define the various cases of noncontiguous I/O and briefly discuss the current noncontiguous I/O methods we will test. In Section 3, we explain the three major I/O characteristics which affect overall I/O performance. In Section 4, we describe the NCIO benchmark, discuss our software stack and present a brief look at our implementation and optimizations. In Section 5, we present the results of the NCIO benchmark and detailed performance analysis. In Section 6, we describe our guidelines for improving I/O performance for scientific application developers. In Section 7, we summarize the work of this paper and discuss possibilities for future work.

2. Noncontiguous I/O: Definition and Methods

All types of I/O access patterns, including both contiguous and noncontiguous cases, are shown in Figure 1. For the contiguous in memory and contiguous in file case, we use the notation c-c. We refer to the noncontiguous cases as nc-c for noncontiguous in memory and contiguous in file, c-nc for contiguous in memory and noncontiguous in file, and nc-nc for noncontiguous in memory and noncontiguous in file.

MPI-IO provides a very rich interface for describing structured noncontiguous data access. In order to efficiently service noncontiguous data access, a variety of I/O methods have been proposed and implemented in MPI-IO. We present a short overview of these methods in the following subsections. A more detailed analysis of each I/O method can be found in [4].

2.1. POSIX I/O

Generally, all parallel file systems support what is called POSIX I/O, which relies on an offset and a length in both memory and file to service an I/O request. This method can service noncontiguous I/O access patterns by dividing up a noncontiguous I/O access pattern into contiguous regions and then individually servicing these contiguous regions with corresponding POSIX I/O operations. The division of the I/O access pattern into smaller contiguous regions significantly increases the amount of I/O requests processed by the underlying file system. Also, the division often forces more I/O requests than actual number of noncontiguous regions in the access pattern as shown in Figure 2a. Serious overhead incurred by servicing so many individual I/O requests limits performance for noncontiguous I/O when using the POSIX interface. Fortunately, for users which have access to file systems supporting only the POSIX interface, two important optimizations were developed for more efficiently performing noncontiguous I/O while using only the POSIX I/O interface: data sieving I/O and two phase I/O. Since our test platform (PVFS2) has no I/O concurrency control we cannot test and do not discuss data sieving I/O (which is described in full in [14]). Even if it was supported, data sieving I/O most likely would provide poor performance since all I/O requests are serialized in aggregate overlapping I/O access patterns (as used in our tests).

2.2. Two Phase I/O

Figure 2c illustrates the two phase method [15], which uses both POSIX I/O and data sieving. The two phase method identifies a subset of the application

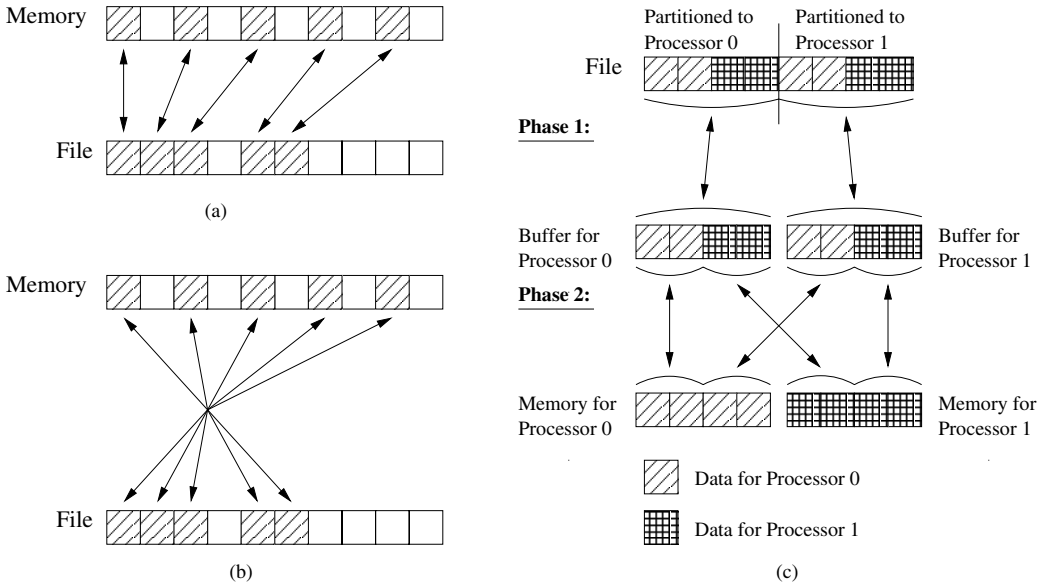

Figure 2. (a) Example POSIX I/O request. Using traditional POSIX interfaces for this access pattern cost five I/O requests, one per contiguous region. (b) Example list I/O or datatype I/O request. Only a single I/O request is necessary to handle this noncontiguous access due to more descriptive I/O requests. (c) Example two phase I/O request. Interleaved file access patterns can be effectively accessed in larger file I/O operations with the two phase method.

processes that will actually do I/O; these processes are called aggregators. Each aggregator is responsible for I/O to a specific and disjoint portion of the file. When performing a read operation, aggregators first read a contiguous region containing desired data from storage and put this data in a local temporary buffer. Next, data is redistributed from these temporary buffers to the final destination processes. Write operations are performed in a similar manner. First, data is gathered from all processes into temporary buffers on aggregators. Next, this data is written back to storage using POSIX I/O operations. An approach similar to data sieving is used to optimize this write back to storage when there are still gaps in the data. As mentioned earlier, data sieving is also used in the read case.

A big advantage of two phase I/O is its ability for I/O aggregators to combine the noncontiguous file descriptions from all processes and perform only a few large I/O operations. One significant disadvantage of two phase I/O is that all processes must synchronize on the open, set view, read, and write calls. Synchronizing across large numbers of processes with different sized workloads can be a large overhead. Two phase I/O performance relies heavily on the MPI implementation's high performance data movement. If the MPI implementation is not significantly faster than the aggregate I/O bandwidth in the system, the overhead of the additional data movement in two phase I/O is likely to prevent two phase I/O from outperforming the direct access optimizations (list I/O and datatype I/O).

2.3. List I/O

The list I/O interface is an enhanced parallel file system interface designed to support noncontiguous accesses shown in Figure 2b. List I/O is an interface for describing accesses that are both noncontiguous in memory and file in a single I/O request by using offset-length pairs. With this interface an MPI-IO implementation can flatten the memory and file datatypes (convert them into lists of contiguous regions) and then describe an MPI-IO operation with a single list I/O request. In previous work [3] we discussed the implementation of list I/O in PVFS and support for list I/O under the ROMIO MPI-IO implementation. The major drawbacks of list I/O are the creation and processing of these large lists and the transmission of the file offset-length pairs from client to server in the parallel file system. Additionally, since we want to bound the size of the list I/O requests going over the network, only a fixed number of file regions can be described in one request. So while list I/O significantly reduces the

number of I/O operations (in our implementation by a factor of 64), a linear relationship still exists between the number of noncontiguous regions and the number of actual list I/O requests (within the file system layer).

2.4. Datatype I/O

Datatype I/O, also illustrated in Figure 2b, is an effort to address the deficiency seen in the list I/O interface when faced with an access that is made up of many small regions, particularly one that exhibits some degree of regularity. Datatype I/O borrows from the datatype concept used in both message passing and I/O for MPI applications. The constructors used in MPI types allow for concise descriptions of the regular, noncontiguous data patterns seen in many scientific applications (such as extracting a row from a two-dimensional dataset). The datatype I/O interface replaces the lists of I/O regions in the list I/O interface with an address, count, and datatype for memory, and a displacement, datatype, and offset into datatype for file. These parameters correspond directly to the address, count, datatype, and offset into the file view passed into an MPI-IO call and the displacement and file view datatype previously defined for the file. The datatype I/O interface is not meant to be used by application programmers; it is an interface specifically for use by I/O library developers. Helper routines are used to convert MPI types into the format used by the datatype I/O functions.

Since it can map directly from an MPI-IO I/O operation with a one-to-one correspondence, datatype I/O greatly reduces the amount of I/O requests necessary to service a structured noncontiguous request when compared to the other noncontiguous access methods. Datatype I/O is unique in comparison with the other methods in that increasing the number of noncontiguous regions that are regularly occurring does not incur any additional I/O access pattern description data passed over the network. When presented with an access pattern of no regularity, datatype I/O breaks down into list I/O.

3. I/O Characteristics Discussion

There are three major I/O access pattern characteristics that seriously affect noncontiguous I/O performance:

- **Region Count** - For some methods, this can cause an increase in the amount of data sent from the clients to the I/O system over the network. When using POSIX I/O, for example, increasing the region count increases the number of I/O requests necessary to service a noncontiguous I/O call. However, some other methods such as datatype I/O are generally expected to be unaffected by this parameter since increasing the region count does not change the size of the access pattern representation for that method in structured data access.
- **Region Size** - Due to the mechanical nature of hard drives, a larger region size will achieve better bandwidth for methods that read/write only the necessary data. Two phase I/O is not likely to improve as much as the individual I/O methods when increasing region sizes since it uses the data sieving optimization. Since memory does not exhibit the same properties as disk, we do no expect any performance change due to larger region sizes in memory.
- **Region Spacing** - If the distance between file regions is small, two phase I/O will improve performance due to internal data sieving. If the distance is small enough, we expect file system block operations may help with caching. We note that spacing between regions is usually different in memory and in file due to the interleaved data operation that is commonly seen in scientific datasets that are accessed by multiple processes. For example, in the FLASH code, the memory structure of the block is different that the file structure, since the file structure takes into account multiple processes.

These characteristics, illustrated in Figure 3, have a different effect on performance when regarding memory access descriptions or file access descriptions.

4. NCIO Benchmark

We have designed an I/O benchmark, *Noncontiguous I/O Test* (NCIO), for studying I/O performance using various I/O methods, I/O characteristics, and noncontiguous I/O cases. We test all three I/O characteristics (region size, region count, and region spacing) against four I/O methods (POSIX I/O, list I/O, two phase I/O, and datatype I/O) in all four of the I/O access cases (c-c, nc-c, c-nc, and nc-nc).

We chose to call MPI_File_sync() after every I/O operation. This enables us to include the time to move the data to the hard drive and not simply test network bandwidth. When using two phase I/O, an optimization in the PVFS2 driver of ROMIO forces only one of the processes to actually call MPI_File_sync(), instead of all 64 processes calling MPI_File_sync() when using individual I/O methods. Test runs that did not

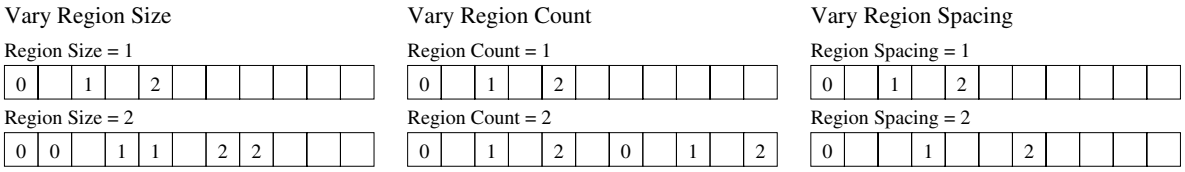

Figure 3. An example of how an access pattern is created from the NCIO parameters.

synchronize the data to disk showed I/O bandwidth as high as 1.59 GBytes / sec. Our best synchronized results will barely reach 24% of that (roughly 389 MBytes / sec). However, when checkpointing and doing other persistent data storage operations, file synchronization to hard disk is not just advised, it is essential to ensure that the data is available when the application is restarted or visualized. All tests used 64 compute processes on 32 dual CPU computers and 16 computers each running a single PVFS2 server process. I/O tests have a lot of variance, so for each data point for we averaged 6 repetitions without the high and the low. We also inserted a one second delay between test runs to try to minimize effects from previous runs.

In order to observe the individual effect of varying each of the I/O characteristics, we hold all other characteristics constant during our experiments. We chose a default region size of 8 bytes to match the size of a double on most computing platforms. This makes the test as I/O intensive as possible and provides a worst case scenario for scientific computing. We used a default spacing of 128 bytes since this maps well to applications that use 16 variables per cell (similar to 17 in IPARS and 24 in the ASC FLASH code). We chose a default region count of 4096 to represent 4096 cells, a mid-size grid. We note that whenever memory and/or file descriptions are contiguous, we used a contiguous MPI datatype for data representation. We used a vector MPI datatype for noncontiguous data representation. While many scientific applications store data in multi-dimensional datasets, which can be represented by vector of vector MPI datatype, we chose to use a single MPI vector datatype to keep consistent region spacing. Our results can be extrapolated to approximately determine multi-dimensional region spacing performance.

4.1. PVFS2 and ROMIO MPI-IO

The Parallel Virtual File System 2 (PVFS2) [16] is a parallel file system for commodity Linux clusters that is a complete redesign of PVFS1 [2]. It provides both a cluster-wide consistent name space and user-defined file striping found in PVFS1, but also adds functionality to provide better scalability and performance.

ROMIO is the MPI-IO implementation developed at Argonne National Laboratory [14]. It builds upon the MPI-1 message passing operations and supports many underlying file systems through the use of an abstract device interface for I/O (ADIO). ADIO allows the use of file-system specific optimizations such as the list I/O and datatype I/O interfaces described here. Additionally, ROMIO implements the data sieving and two phase optimizations as generic functions that can be used for all file systems supported by ADIO.

4.2. Implementation and Optimizations

In order to test each noncontiguous I/O method in PVFS2, we had to implement them in the PVFS2 driver of ROMIO. We rewrote the list I/O implementation to simplify the code and fix bugs. We added support for datatype I/O by breaking down MPI datatypes into PVFS2 datatypes to produce native PVFS2 I/O datatype requests. We changes the ADIO PVFS2 hints to select different I/O methods with the MPI_Info_set() call. This work is quite substantial, however, due to a shortage of space, we cannot detail our implementation. Our modified PVFS2 driver has been sent to ROMIO developers.

We rewrote the I/O portion of the PVFS2 server trove component to use read(), write() and lseek() calls instead of lio_listio(). We found this optimization improved performance over an order of magnitude in some noncontiguous I/O cases. We will send this patch to the PVFS2 developers when it stabilizes.

5. Performance Evaluation

All tests were run on the Feynman cluster at Sandia National Laboratories. Feynman, composed of Europa nodes, Ganymede nodes, and I/O nodes, has a total of 371 computers. In order to keep our testing as homogeneous as possible, we only used the Europa nodes. The Europa nodes are dual 2.0 GHz Pentium-4 Xeon CPUs with 1 GB RDRAM. They are connected with a Myrinet-2000 network and use the Redhat Linux En-

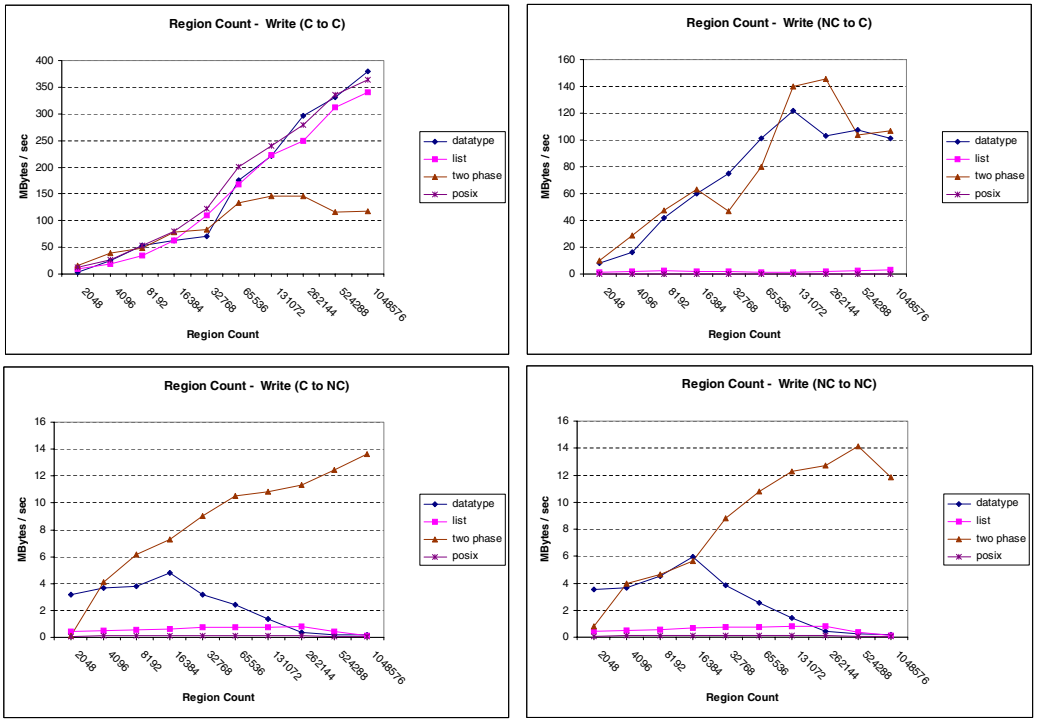

Figure 4. NCIO results from testing various region counts.

terprise operating system. Since each computer has dual CPUs, we used 2 compute processes per node.

We used 16 computers for our PVFS2 file system. All 16 computers ran the PVFS2 server with one computer additionally handling metadata server responsibilities. All PVFS2 files were created with the default 64 KByte strip size, totaling to a 1 MByte stripe across all I/O servers. Figure 4 shows our results when varying the region count. Figure 5 shows our results when varying the region size. Figure 6 shows our results when varying the region spacing.

5.1. NCIO - Vary Region Count Results

In the c-c case, increasing the region count increases overall bandwidth. The data sizes per process range from 16 KBytes at the low end (region count = 2048) to 8 MBytes at the high end (region count = 1048576). The overhead of a double network transfer makes two phase I/O much less efficient than the other methods.

In the nc-c case, two phase I/O and datatype I/O perform fairly well as the impact of varying the region count with noncontiguous memory access isn't as good as c-c, but still scales up some. List I/O performance immediately drops since only 64 offset-length pairs are transferred per request. List I/O must process 1048576 / 64 = 16384 requests per process, where each request is only 64 * 8 = 512 bytes. Such a small size is not suited for a hard disk. Datatype I/O drops significantly as well due to the PVFS2 server side storage implementation. The PVFS2 flow component can only handle processing 1024 offset-length pairs at a time on the server side. Therefore, datatype I/O can only write 8 KByte regions at a time, which is better than the 512 byte regions in list I/O but still not large enough to achieve high disk bandwidth.

In the c-nc case, we note that I/O bandwidth is less than 5% of the maximum bandwidth we saw in c-c. Noncontiguous data access in file is really hard on the individual I/O methods (POSIX I/O, list I/O, and datatype I/O). Performance of datatype I/O drops off at about 16384 regions. This is mostly likely caused by the fact that datatype I/O requests block the PVFS2 server until they completely finish. Fsync() calls immediately come after the datatype I/O requests and block the PVFS2 server from doing other client I/O requests. Since the fsync() calls arrive at intervals after each datatype I/O request is completed the aggregate write I/O pattern will have 64 I/O requests interrupted with intermittent fsync() calls. List I/O performance isn't as affected by this fsync() problem because it breaks down into I/O requests of 64 file offset-length

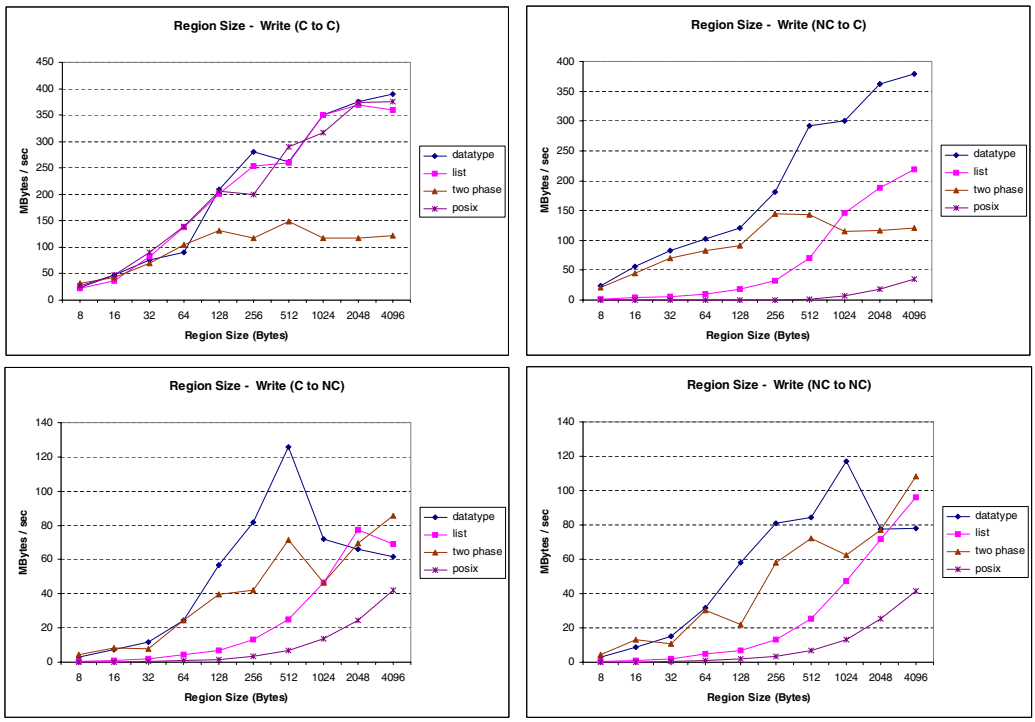

Figure 5. NCIO results from testing various region sizes.

pairs. Therefore, the list I/O requests all finish about the same time (most likely around when the other list I/O requests are around their last 64 file offset-lengths pairs) and then the fsync() calls occur, making it less costly to fsync() versus the datatype I/O case. The MPI_File_sync() time for the list I/O test in the c-nc case never exceeds 18 seconds when region count = 1048576. The datatype I/O MPI_File_sync() time for the c-nc case reaches as high 2061 seconds. POSIX I/O is so slow that with the region count = 1048576, it took 11014.66 second (0.047 MBytes / sec) to complete a repetition. In fact, two phase I/O is performing quite well in comparison because it only does a single fsync() call between all the processes. This allows it to scale up much better than the other methods.

In the nc-nc case, performance is nearly identical to the c-nc case. If the file description is noncontiguous then making the memory description noncontiguous has little impact.

5.2. NCIO - Vary Region Size Results

In the c-c case, we see the same trends as the c-c case of the region count test. Increasing the total data size from 32 KBytes (region size = 8) to 16 MBytes (region size = 4096) / process scales up the I/O bandwidth. Since we have 16 I/O servers, each server only receives about 1 MByte per compute process at the maximum size. With larger sizes we could certainly achieve higher I/O bandwidth. Two phase I/O tails off here again due to its double network transfer.

In the nc-c case, datatype I/O performs almost equivalent to the c-c case. Since the default region count = 4096, and there are 16 I/O servers, the flow component isn't a bottleneck for datatype I/O. Two phase I/O performance is nearly identical to its nc-c case of the region count test. List I/O actually does fairly well here as well since it isn't as limited by its 64 offset-length pair maximum as it was in the nc-c case of the region count test.

In the c-nc case, datatype I/O peaks rapidly and then falls due to the fsync() issue discussed in the c-c case of the region count test. If the fsync() calls happened after all the writes from all the processes finished we would expect the scaling to continue. All I/O methods benefit from the increased region sizes. Even two phase I/O can benefit quite a bit from the larger region sizes since it doesn't have to pass around such a large amount of offset-length pairs to each of the aggregators and its percentage of useful data acquired while using the data sieving method is improving.

In the nc-nc case, the trends of match the c-nc case,

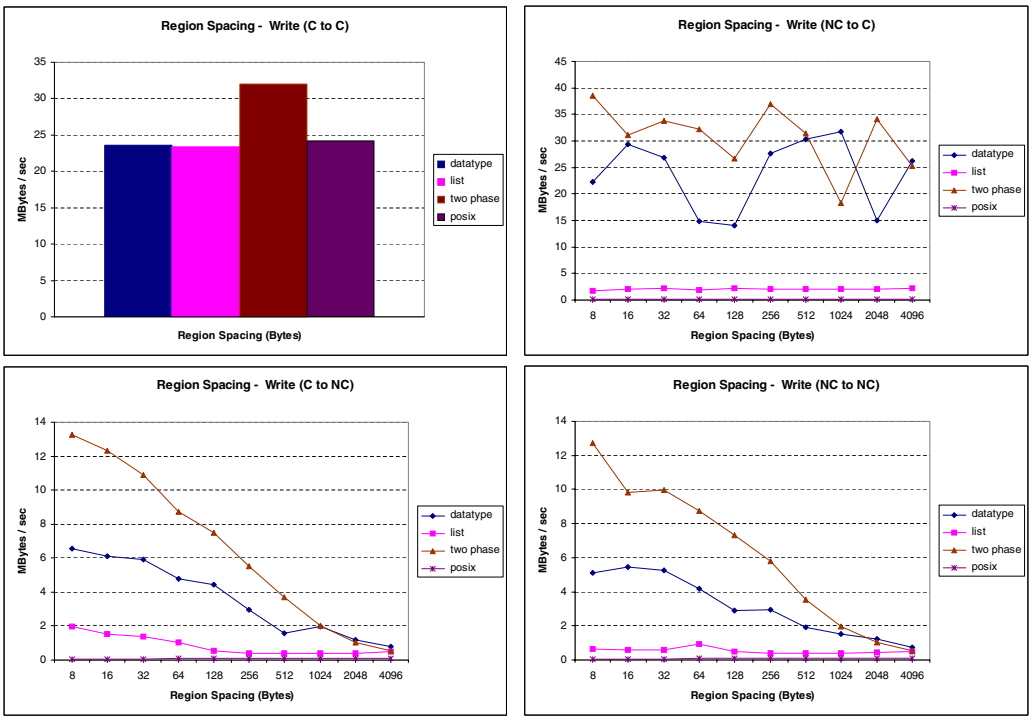

Figure 6. NCIO results from testing various region spacing.

again showing that if the file description is noncontiguous, then it makes little performance difference if the memory description is contiguous or noncontiguous.

5.3. NCIO - Vary Region Spacing Results

In the c-c case, we only have a single bar since varying region spacing has no effect on a contiguous memory or file description. One interesting thing is that two phase I/O performs slightly better than the other methods. This is also evident in the c-c cases for the other tests when each process is writing 32 KBytes. This is a case where the MPI_File_sync() costs are cheaper for two phase I/O (due to the two phase MPI_File_sync() optimization). The average MPI_File_sync() cost for the two phase I/O runs was 0.0391 seconds versus 0.0705 seconds for the normal case, which allows two phase I/O a win at this small write case.

In the nc-c case, datatype I/O and two phase I/O fluctuate (even with the average policy we used with 6 repetitions and eliminating the high and the low). The main cause of the fluctuation is that the writes are only 32 KBytes per process and can only fill half a strip on the I/O servers. If we average out the datatype I/O numbers, the result is 23.83 MBytes / sec (23.65 MBytes / sec on the c-c case). If we average out the two phase I/O numbers, the result is 30.832 MBytes / sec (32.017 MBytes / sec in the c-c case). When going from the c-c case to the nc-c case, datatype I/O and two phase I/O retain their performance. However, list I/O takes a large drop due to only processing 64 offset-length pairs at a time. Therefore, each process calling list I/O must do a series of small writes (64 * 8 = 512 bytes) to finish the access pattern. POSIX I/O is making 4096 writes of size = 8 bytes, which leads to an aggregate bandwidth of no more than 0.145 MBytes / sec.

In the c-nc case, performance drops for all I/O methods quite significantly. The individual I/O methods are all suffering from disk seek penalties. Writing 8 bytes and then skipping up to 4096 bytes makes performance drop rapidly. Two phase I/O surpasses the other methods due to its internal data sieving implementation making larger I/O operations until about 1 KByte spacing, where it appears that the penalty of data sieving large holes overwhelms the benefits of larger I/O operations and allows datatype I/O to surpass it.

The nc-nc case shows a similar trend to the c-nc case. Again we note that moving from c-nc to nc-nc makes only a small performance difference.

6. I/O Guidelines

- **Scientific applications accessing structured datasets should use the MPI-IO interface.** If an application is not using MPI-IO, then unless the application is doing its own optimization, its performance most likely follows the POSIX I/O results. For example, when region count = 4096 in the nc-nc region count test, switching from POSIX I/O to two phase I/O would improve aggregate bandwidth by a factor of about 260.

- **When using individual I/O methods, choose datatype I/O.** In nearly all cases datatype I/O exceeded the performance of the other individual I/O methods. When region count = 131072 in the nc-c region count test, datatype I/O outperformed list I/O by a factor of 90 and POSIX I/O by a factor of over 800.

- **Try to group writes into larger file regions as much as possible.** Larger file regions favor better performance. When the file description is noncontiguous, all I/O methods performed best on the file region test with large file region sizes. For example, if we have a multi-dimensional dataset with several variables per cell where the variables are the lowest array dimension when flattened into a file, try to write as many of the variables at a time is possible (which should increase file region sizes). Also, in general, nc-c tests outperformed c-nc and nc-nc tests. Performance generally improved between a factor of 2 to 4 for list I/O, datatype I/O, and two phase I/O when moving from c-nc to nc-c in the region size test,

- **Reduce the spacing between file regions being written.** All I/O methods, while writing the same amount of data with the same region sizes decreased significantly in performance as the file spacing increased. Application designers can hopefully alleviate file region spacing issues by helping control which processes are responsible for portions of the data grid and keep them logically close in file. In two phase I/O c-nc case, when going from a region spacing of 1024 to 512, performance increased by about 85%.

- **Noncontiguous memory description versus contiguous memory description makes little difference if the file description is noncontiguous.** If the file description is noncontiguous, it may be tempting to copy noncontiguous memory data into a contiguous buffer before using a MPI_Write() call, however, our results show that this won't affect performance and will just incur extra memory overhead.

- **If the file description is noncontiguous, the region sizes are small and the count of file regions is very large (i.e. greater than 16384), use two phase I/O.** Two phase I/O, with its aggregate data sieving properties, is best suited for this case since it creates large I/O requests for the hard disk. However, at larger file region sizes, the individual I/O methods may surpass the two phase I/O performance quite rapidly.

- **Consider the cost of two phase I/O synchronization.** While two phase I/O performance may look appealing in some cases, one caveat with two phase I/O is that all processes must synchronize on open, read/write, and setting the file view. If your application automatically synchronizes before all compute processes write, the cost of two phase I/O synchronization will be minimal. However, if computational times between processes vary (i.e. an AMR code) and data is written at random intervals, then even if two phase I/O results in this paper may have surpassed individual I/O methods, the penalty of waiting may take away any benefit of increased bandwidth of the actual I/O operation.

7. Conclusions and Future Work

We have described the problem of storing structured scientific data and how it is affected by I/O characteristics and I/O methods. Our contributions include:

- **MPI-IO, PVFS2 implementation and tuning** - We rewrote the I/O portion of the ROMIO PVFS2 device driver to support list I/O and datatype I/O. We reimplemented the read/write procedures in the PVFS2 trove component to improve performance by over an order of magnitude for noncontiguous access patterns in many cases.

- **Created NCIO benchmark and provided detailed performance evaluation** - We implemented the NCIO benchmark to provide a useful tool for learning more about how noncontiguous I/O. Our performance analysis helps to understand how and why I/O access patterns are affected by various I/O methods, I/O cases, and I/O characteristics.

- **Provided guidelines for application developers** - Our results and discussion have led us to produce a series of insights for how and when application designers can get the best utilization out of

their high performance file system. Many of these suggestions can make performance differences of 1 to 2 orders of magnitude.

While our testing does not cover the entire range of possibilities of how one create access patterns, it has generated a simple set of guidelines that can aid in structured scientific storage and make significant I/O improvements.

There are a range of possibilities for future work. We plan to improve the two phase I/O method in ROMIO. We will experiment with two phase I/O to internally use the individual I/O methods, which may improve performance for certain cases. We can work on the request scheduler on the server side to make sure to delay requested fsync() calls until pending I/O operations to a particular file complete to reduce client MPI_File_sync() costs.

While noncontiguous file access performance is less than ideal, we feel that the I/O methods we have available provide a substantial improvement over the POSIX I/O interface. The noncontiguous file access situation has improved significantly over the last decade and our paper provides some insight on how to take advantage of these improvements. In summary, we hope our study will benefit scientific application designers in more efficiently storing their structured datasets.

Acknowledgments

We would like to thank the PVFS2 development team (Rob Ross, Walter Ligon, Phil Carns, Rob Latham, and several others) for helping to debug the datatype, flow, and trove components of PVFS2.

This work was supported in part by Sandia National Laboratories and DOE under Contract 28264, DOE's SciDAC program (Scientific Data Management Center) award number DE-FC02-01ER25485, the NSF/DARPA ST-HEC program under grant CCF-0444405, and the DOE HPCSF program.

References

[1] S. J. Baylor and C. E. Wu. Parallel I/O workload characteristics using Vesta. In *Proceedings of the IPPS '95 Workshop on Input/Output in Parallel and Distributed Systems*, pages 16–29, April 1995.

[2] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A parallel file system for Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, Atlanta, GA, October 2000. USENIX Association.

[3] A. Ching, A. Choudhary, K. Coloma, W. keng Liao, R. Ross, and W. Gropp. Noncontiguous access through MPI-IO. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

[4] A. Ching, A. Choudhary, W. keng Liao, R. Ross, and W. Gropp. Evaluating structured I/O methods for parallel file systems. In *International Journal of High Performance Computing and Networking*.

[5] P. F. Corbett and D. G. Feitelson. The Vesta parallel file system. *ACM Transactions on Computer Systems*, 14(3):225–264, August 1996.

[6] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[7] B. Fryxell, K. Olson, P. Ricker, F. X. Timmes, M. Zingale, D. Q. Lamb, P. MacNeice, R. Rosner, and H. Tufo. FLASH: An adaptive mesh hydrodynamics code for modeling astrophysical thermonuclear flashes. *Astrophysical Journal Supplement*, 131:273, 2000.

[8] HDF5 home page. http://hdf.ncsa.uiuc.edu/HDF5/.

[9] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Sigel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific I/O interface. In *Proceedings of Supercomputing 2003*, November 2003.

[10] N. Nieuwejaar and D. Kotz. The Galley parallel file system. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[11] M. L. Norman, J. Shalf, S. Levy, and G. Daues. Diving deep: Data-management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science and Engg.*, 1(4):36–47, 1999.

[12] J. Saltz, U. Catalyurek, T. Kurc, M. Gray, S. Hastings, S. Langella, S. Narayanan, R. Martino, S. Bryant, M. Peszynska, M. Wheeler, A. Sussman, M. Beynon, C. Hansen, D. Stredney, and D. Sessanna. Driving scientific applications by data in distributed environments. In *Workshop on Dynamic Data-Driven Application Systems*, June 2003.

[13] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, San Jose, CA, January 2002. IBM Almaden Research Center.

[14] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189. IEEE Computer Society Press, February 1999.

[15] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 23–32, May 1999.

[16] The parallel virtual file system 2 (PVFS2). http://www.pvfs.org/pvfs2/.