Guidelines for Efficient Parallel I/O on the Cray XT3/XT4

Jeff Larkin, Cray Inc. and *Mark Fahey*, Oak Ridge National Laboratory

ABSTRACT: This paper will present an overview of I/O methods on Cray XT3/XT4 supercomputers. It will show several benchmark results and interpret those results to propose guidelines for maintaining efficient I/O rates on a Lustre filesystem on a Cray XT3/XT4. Finally it will show results from an application implementing these guidelines and describe possible future investigations.

KEYWORDS: CRAY XT3/XT4, PARALLEL IO, IOR, LUSTRE, BENCHMARK, MPI-IO, PERFORMANCE

1 Introduction

There has been a clear trend in recent years towards increasingly larger-scale supercomputers. One can examine the historical data provided by the Top 500 List [TOP500] to find a wealth of evidence to support this claim. As microprocessor makers move their products to multi-core processors in order to sustain Moore's Law, the number of processor cores in even the smallest supercomputers will begin to seem *massive* by today's standards. Researchers are currently scrambling to determine how to scale their algorithms to machines with tens or hundreds of thousands of cores, but computational performance is only one of the challenges they will face at this scale. The days when I/O could be treated as an after-thought to algorithmic performance are coming to an end. It is important that application developers begin to examine the I/O capabilities that will be available to them and how to best utilize them.

In this paper we will give an overview of the I/O subsystem on Cray XT3/XT4 computers. We will then show and interpret data to evaluate the current state of I/O on a large-scale, Cray XT3/XT4 system and provide guidelines for efficient I/O on large, Cray XT3/XT4 systems.

2 An overview of Lustre on a Cray XT3/XT4

When high bandwidth I/O operations are required on an XT3/XT4 system, they should be done within a Lustre parallel filesystem [LUSTRE]. Cray has partnered with Cluster Filesystems, Inc (CFS) [CFS] to develop a client for Lustre on the XT3/XT4 in the form of liblustre.

2.1 Lustre Basics

Lustre is a clustered filesystem designed to provide large, high bandwidth storage on large, clustered computers. Figure 1 depicts how the lustre architecture is connected to compute nodes of a Cray XT3/XT4 system.

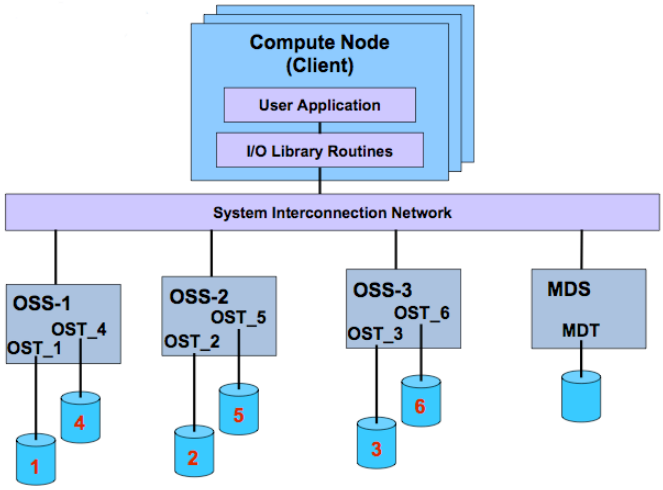

Figure 1 Lustre architectural diagram.

At its lowest level, the Lustre filesystem (LFS) has two basic types of servers: the Metadata Server (MDS), and Object Storage Servers (OSS). As the name implies, the MDS is a database that holds the file metadata for the entire filesystem. Whenever a metadata operation occurs, such as an open, the client must poll the MDS. At the time of writing this paper, Lustre filesystems are limited to one MDS. A Lustre filesystem may have one or more OSSes, which handle storing data to disk. Each OSS has one or more Object Storage Targets (OST), where the file data actually resides. It is not uncommon for a single OSS to

serve several OSTs. Files are broken into objects, which are stored on OSTs in a round-robin fashion.

A user's files are striped over one or more OSTs when they are written to the filesystem. The user has the ability to adjust the number of stripes (stripe width) and the size of the objects on disk (stripe size). To put it more plainly: stripe width relates to how many OSTs are used to store a file and the stripe size relates to how large an object is on disk. Files inherit these parameters from their parent directory and users may adjust these parameters using the *lfs* command. A file cannot have a stripe width greater than the total number of OSTs configured in the host filesystem. At publication time, the total number of OSTs for a Lustre filesystem is limited to 160.

2.2 Lustre on Jaguar

Over the course of collecting data for this paper, the Lustre filesystem on the Cray XT3/XT4 system at Oak Ridge National Lab ("Jaguar") underwent reconfiguration. Early benchmark results were collected on a filesystem with 160 OSTs, broken over 40 OSSes. More recent results are taken from a slightly smaller filesystem, configured with 144 OSTs on 36 OSSes. The actual filesystem configuration will be explicitly given for each of the benchmark results below.

The current system configured with 144 OSTs uses 36 OSSes. Each OSS has two 4 gigabit fibre channel cards. Each card serves two OSTs. The setup uses two "tiers" per logical unit number (LUN). A tier is a DDN term that refers to a raid group. Each tier has 9 disks configured as a 8+1 RAID5. So each LUN has 18 disks behind it. The drives are 300 GB 10K RPM fibre channel drives. A LUN is zoned to specific ports, which correspond to specific OSSes. Aggregate peak bandwidth is 144*4 Gb/s = 72 GB/s.

3 Benchmarking Methodology and Results

In this section three codes are used to test various aspects of I/O performance of the lustre filesystem described in section 2. As indicated above, results were obtained with different configurations of the filesystem: namely when configured with 160 OSTs and then again with 144 OSTs.

3.1 Code 1

Initial benchmarking was done using a custom code [CUSTOM1] designed to emulate writing a large amount of data to disk from all processors. This is a very simple code that opens an MPI file across the processors and performs buffered writes using the `mpi_file_write_at` method to write to a shared file at given offsets. This, of course, assumes that a regular amount of data is written by each processor, which makes the calculation of offsets and distribution of I/O operations trivial. The benchmark varies the number of writers, the size of a user-allocated buffer for each writer, the size of the Lustre stripe, and the number of Lustre stripes. In its original form, this benchmark opened the file over a subset of the total processors and uses MPI operations to communicate the necessary data to these processors for writing. It was determined early in testing that the high bandwidth SeaStar network of the Cray XT3/XT4 makes the communication portion negligible. For benchmarking convenience, the code was rewritten to ignore the subset communication and treat each node as a writer. This allowed for more benchmark runs to be made by reducing the number of processors needed for each run. While subsetting was not used to collect the data presented in this section, it should be assumed when interpreting these results that the writers are a subset of the total processors. Evidence in support of using a subset of processors to achieve better I/O performance will be presented in a later section.

Results in this section were collected on a Cray XT4 running version 1.5 of Unicos/lc. The benchmark was compiled with version 6.2 of the PGI compiler suite. The Lustre filesystem was configured with 160 OSTs for the tests in this section.

3.1.1 Single Stripe Performance

There is an unfortunate misconception that sending all data to a single node to be written to disk will achieve suitable bandwidth when performed on a parallel filesystem. This notion fails to take into account the networking overhead of transferring to the single writer and the bottleneck of attempting to output all of a program's data through the network connection of a single node. Writing from a single node simply will not saturate the available filesystem bandwidth needed for large I/O operations, but can give some insight for tuning parallel I/O operations. For example, Figure 2 illustrates the bandwidth of a single writer to a single OST, varying the user allocation buffer and stripe size.

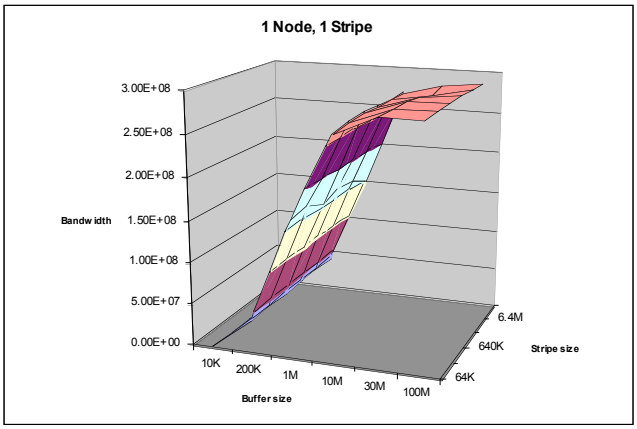

Figure 2: This graph shows bandwidth when writing from one processor to one stripe, varying the size of the buffer and stripe.

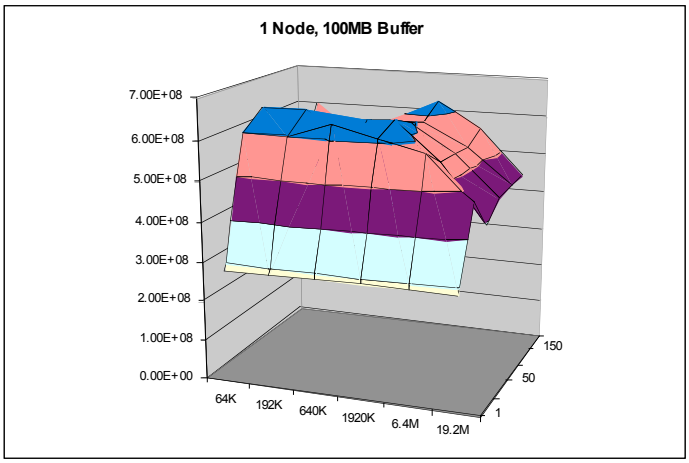

Figure 3a: Bandwidth from one writer with a 100MB buffer, varying the stripe size and width.

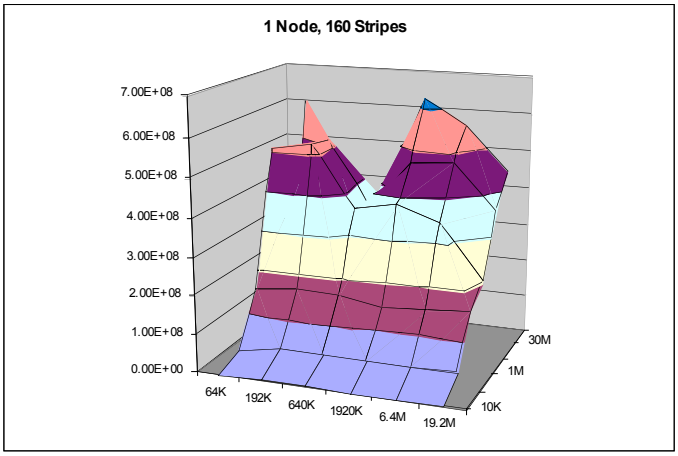

Figure 3b: Bandwidth from 1 writer to a file with a stripe width of 160, varying the size of the buffer and stripes.

It should be clear from the figure that one writer and one OST will not achieve high bandwidth, but the striking observation from this figure is the importance of buffering I/O operations. Observe that varying the size of the filesystem stripe had no real effect on the bandwidth, but varying the size of the user buffer affected the over bandwidth greatly. The need to buffer I/O operations should be obvious and Figure 2 shows that a 1-10MB buffer can significantly improve write performance.

Figures 3a-b show further evidence of the need to use multiple writers to achieve reasonable bandwidth. Observe that no matter how widely the file is striped, this benchmark was unable to achieve greater that 700MB/s of write bandwidth. Although an increase in bandwidth was observed, even with maximum striping the performance was well below acceptable levels. It is simply impossible for a single writer to saturate the available filesystem bandwidth.

3.1.2 Fixed Number of Stripes

By fixing the number of stripes for a given file and varying the number of writers we are able to make observations about the desirable ratio of writers to stripes. While a one-to-one ratio is logical, it may not be the most practical or efficient ratio.

The graphs in Figures 4a-d plot the I/O performance for a fixed strip count of 150 while varying the stripe and buffer sizes along the axes and varying the number of writers between graphs.

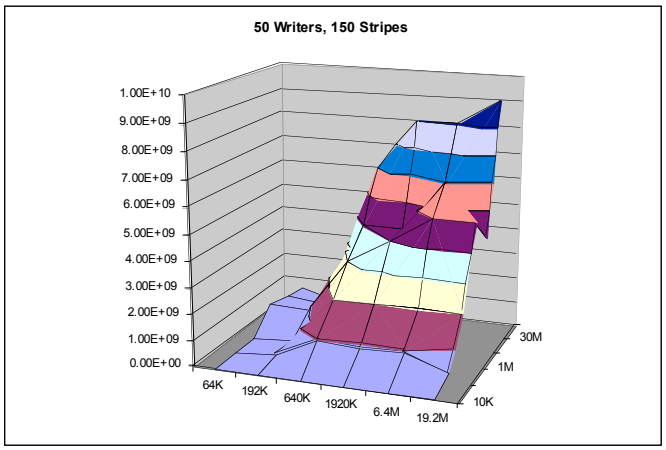

Figure 4a: Write bandwidth with 50 writers and 150 stripes. The bottom and side axes of each graph are the stripe user buffer size, respectively.

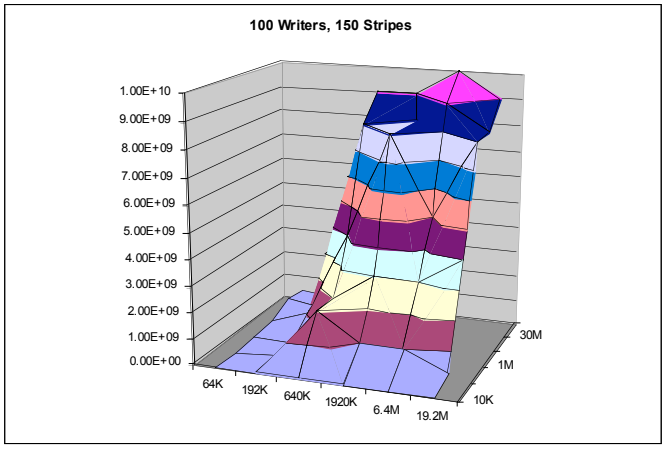

Figure 4b: Write bandwidth with 100 writers and 150 stripes. The bottom and side axes of each graph are the stripe user buffer size, respectively.

The point that should be observed from Figure 4d is that having significantly fewer writers than stripes does result in lower write bandwidth, while having nearly as many or slightly more writers than stripes achieves higher performance. The difference between the achieved bandwidth at 100 writers is insignificant from the bandwidth at 150 or 300 writers.

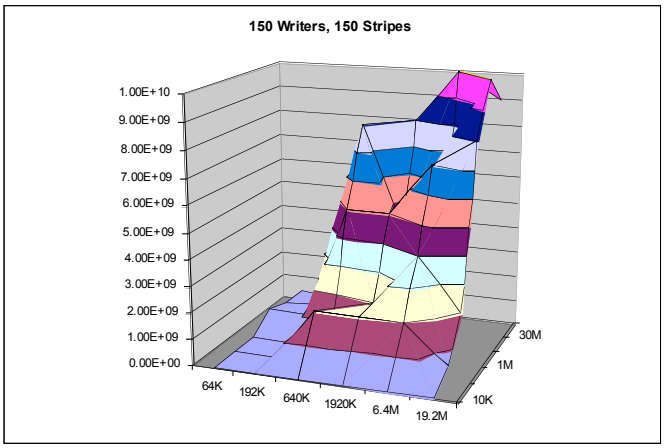

Figure 4c: Write bandwidth with 150 writers and 150 stripes. The bottom and side axes of each graph are the stripe user buffer size, respectively.

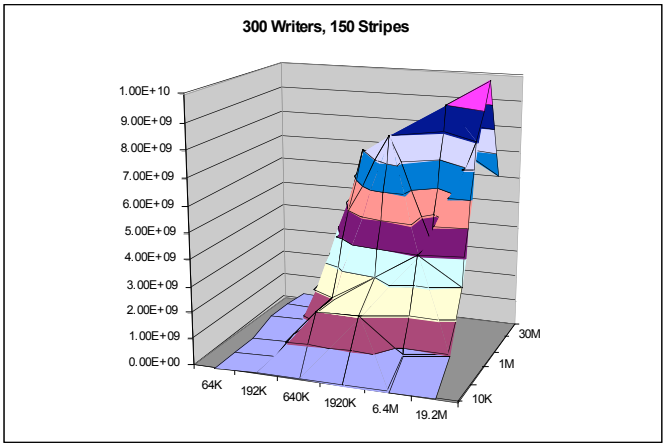

Figure 4d: Write bandwidth with 300 writers and 150 stripes. The bottom and side axes of each graph are the stripe user buffer size, respectively.

These graphs suggest that there is no reason to have more or less than a one-to-one relationship between writers and stripes. Data presented in a later benchmark, however, approaches this question from a different direction and implies that one-to-one may not actually be the correct ratio.

3.1.3 Fixed Number of Writers

Fixing the number of writers and varying the other parameters can also provide interesting insights. The graphs in Figures 5a-d fix the number of writers and emphasize the need for buffering by varying the size of buffers between graphs.

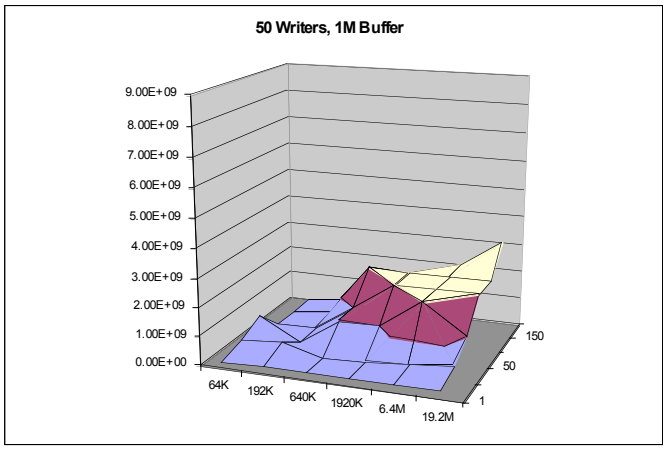

Figure 5a: Write bandwidth with 50 writers and 1 MB buffer. The bottom and side axes are stripe size and count respectively.

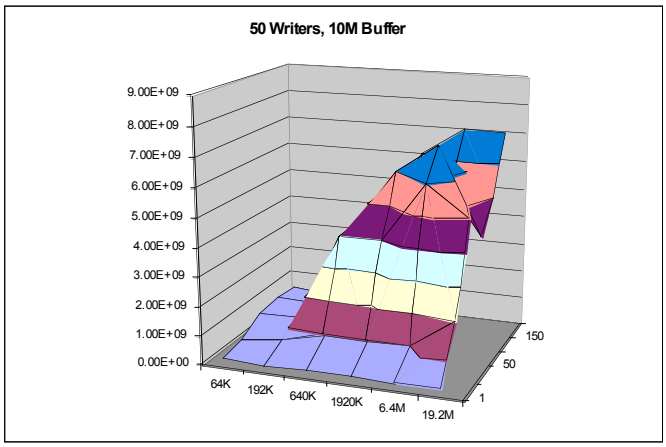

Figure 5b: Write bandwidth with 50 writers and 10 MB buffer. The bottom and side axes are stripe size and count respectively.

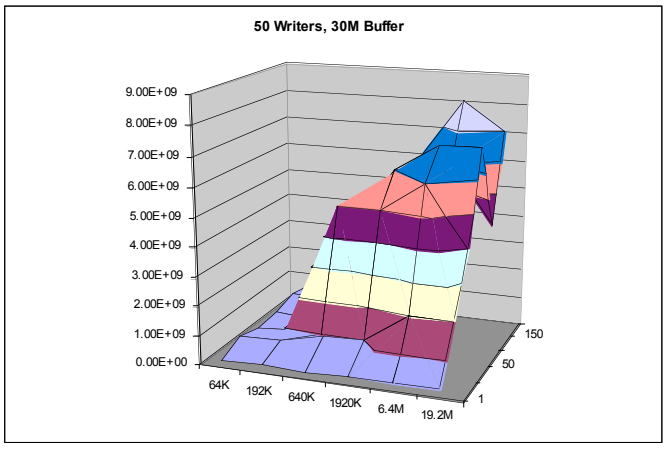

Figure 5c: Write bandwidth with 50 writers and 30 MB buffer. The bottom and side axes are stripe size and count respectively.

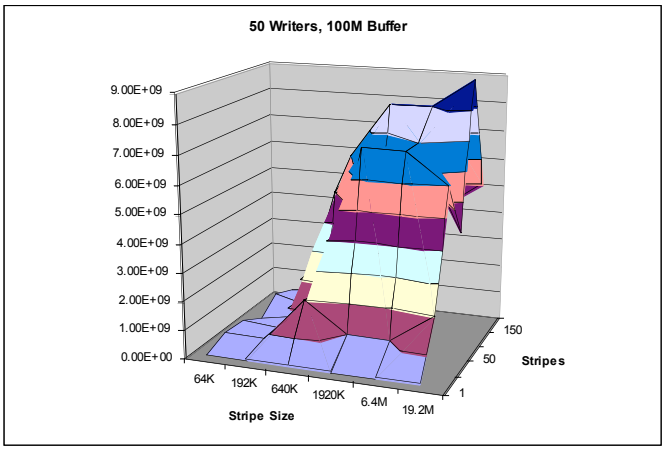

Figure 5d: Write bandwidth with 50 writers and 100 MB buffer. The bottom and side axes are stripe size and count respectively.

One should note the significant improvement in performance as the buffer size increases. Buffer sizes below 1MB are not shown, but are even below the performance of 1MB. Many applications are unable to sacrifice 100MB for use in I/O buffers, but the above graphs should make it clear that whatever sacrifice can be made for I/O buffers will result in improved I/O performance.

3.1.4 Interpretation of results

The clearest result from the above benchmarks is the emphasis of buffering I/O operations. While adjusting the stripe size and width does provide noticeable gains in write performance, the use of large I/O buffers seems to have the most pronounced effect on performance. This benchmark also encourages using at least as many writers as the stripe count, but does not show a benefit from utilizing more writers than stripes.

3.2 Code 2: IOR

IOR (Interleaved Or Random) [IOR] is a parallel file system test code developed by the Scalable I/O Project at Lawrence Livermore National Laboratory. This parallel program performs parallel writes and reads to/from a file using MPI-IO (or optionally POSIX or HDF5) and reports the throughput rates. The name of the program is something of an historical artifact because this version has been simplified to remove the random I/O options. IOR can be used for testing performance of parallel file systems using various interfaces and access patterns. IOR uses MPI for process synchronization.

Results in this section were collected on a Cray XT4 running version 1.5 of Unicos/lc. The benchmark was compiled with version 6.1 of the PGI compiler suite. The Lustre filesystem was configured with 144 OSTs for the tests in this section.

3.2.1 Scaling results

IOR was used to test the scalability of the lustre filesystem by doing parallel I/O tests out to many thousands of processors, Figure 6 shows the performance results when using IOR with constant buffer size per client (core), and increasing the number of clients.

The upper plot in Figure 6 is the case when writing or reading with 1 file per client, while the lower graph is for a shared file. The maximum achieved bandwidths are 42 GB/s (read) and 25 GB/s (write) for one file per client and 34 GB/s (read) and 22 GB/s (write) for a shared file.

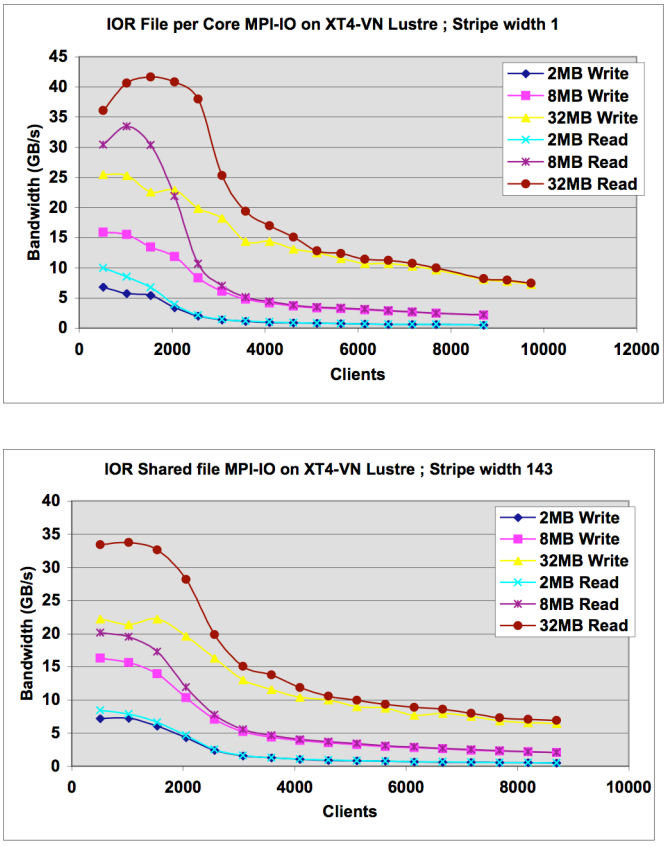

Figure 6: These graphs fix the buffer size per core at 2, 8, or 32MB and vary the number of clients along the x-axis. The y-axis is the achieved bandwidth.

The scalability of lustre was also tested by keeping the aggregate file size constant while increasing the number of clients – an attempt to more accurately simulate what a user of a large-scale XT3/XT4 machine might consider when designing a large run. In Figures 7 and 8, the aggregate size of the file was kept constant at 16 GB and 64 GB, respectively. In other words, as the number of clients increased, the I/O per core decreased. In Figures 7 and 8, the upper plot shows the performance for 1 file per client while the lower depicts the shared-file performance.

In Figure 7, the maximum write performance is approximately 22 GB/s for both file-per-process and shared-file methods. The maximum read performance is approximately 37 GB/s when doing a file-per-process and 33 GB/s for a shared file.

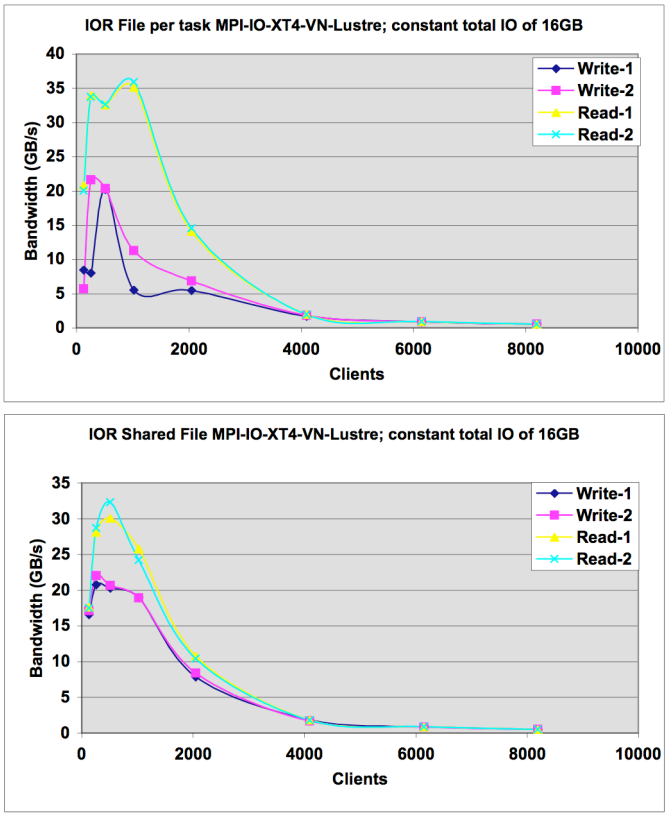

Figure 7: These graphs fix aggregate file at 16 GB and vary the number of clients along the x-axis. The y-axis is the achieved bandwidth.

In Figure 8, the maximum write performance is approximately 26 GB/s for the file-per-process method and 22 GB/s for the shared-file method. The maximum read performance is approximately 40 GB/s when doing a file-per-process and 37 GB/s for a shared file. As expected, I/O is more efficient when writing out larger buffers as shown by the greater I/O bandwidths for the 64 GB file versus the 16 GB file.

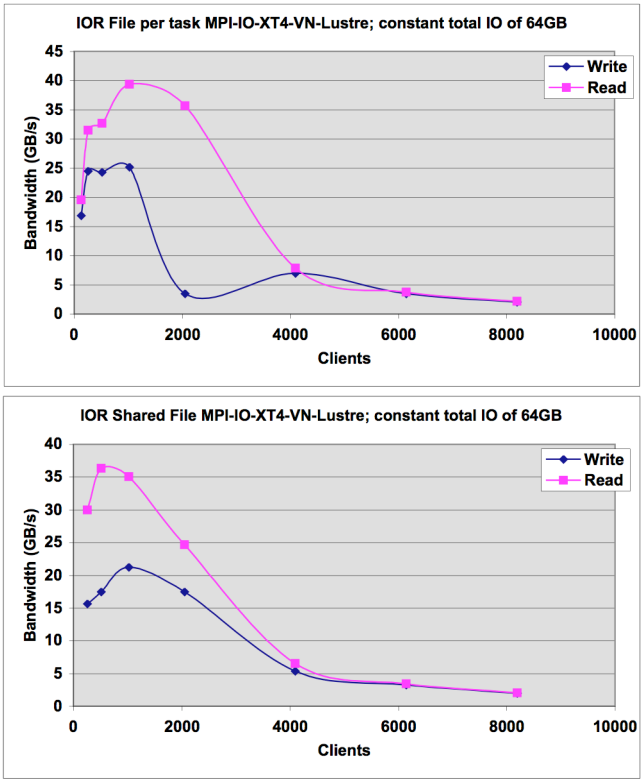

Figure 8: These graphs fix aggregate file at 64 GB and vary the number of clients along the x-axis. The y-axis is the achieved bandwidth.

3.2.2 Interpretation of Results

The results from above clearly illustrate that the achievable bandwidth drops dramatically in all three scenarios for both a file per process and a shared file after it reaches a maximum somewhere around 500-2000 clients. Thus, it seems clear that using a subset of the application tasks to do I/O will result in overall better performance *at scale*. Additionally, these plots indicate that using more than 1 client per OST is the only way to get the practical peak performance. And a third observation is that performing I/O with one file per client with approximately 1K clients is noticeable more efficient than doing I/O to a shared file.

3.3 Code 3

Since the IOR results indicated a subset of clients performing I/O would be more efficient than all clients at scale, another custom I/O code [CUSTOM2] was written to test this hypothesis. This custom Fortran code is designed to write out contiguous buffers by each process either to a single-shared file or alternatively, to a file per process. Furthermore, this code was designed to take in runtime parameters to define a subset of processes with which to do I/O, and, in addition, to use one of several methods of moving the data to the subset of processes to do the I/O. The intent (as with [CUSTOM1]) was to model the behavior of writing a large amount of data to disk, as in a checkpoint.

3.3.1 Subsetting results

Test 1: Results for this test were collected on a Cray XT3 running version 1.5 of Unicos/lc. The benchmark was compiled with version 6.1 of the PGI compiler suite. The Lustre filesystem was configured with 96 OSTs for the tests in this section. First, the code was run with 8640 processes with all processes writing (and then reading) a 5 MB buffer to their own file. This test resulted in an effective bandwidth of 1.4 GB/s for writes and 2.0 GB/s for reads. Then a run was done with a subset of 960 processes out of 8640 aggregating the buffers from 9 tasks and then writing out a 45 MB buffer. For this run, the aggregate bandwidth was 10.1 GB/s for writes and 10.3 GB/s for reads.

Test 2: Results for this test were collected on a Cray XT4 running version 1.5 of Unicos/lc. The benchmark was compiled with version 6.1 of the PGI compiler suite. The Lustre filesystem was configured with 144 OSTs for the tests in this section. This test is very similar to test 1, but with a few changes. The code was run with 9216 processes each with an 8 MB buffer. First, with all processes doing I/O, the aggregate write bandwidth was 0.6 GB/s. Then, with 1024 processes aggregating I/O from 9 processes each to write out a 72 MB buffer, the aggregate bandwidth was 10.4 GB/s.

3.3.2 Interpretation of Results

The results clearly provide evidence that using a subset of nodes for I/O results in significantly better performance than when using all available processes at scale.

4 I/O Guidelines

Based on the data and the corresponding observations presented above, the following guidelines are proposed:

1. Do large I/O. It is recommended to use buffers of at least 1 MB and 10 MB if possible. In some cases one can use IOBUF on the XT3/XT4 platform to buffer output.
2. Do parallel I/O. The only way to take full advantage of the Lustre filesystem is to do parallel I/O.

3. Stripe as close to maximum number of OSTs as possible if writing to a shared file and if you have more clients than OSTs.
4. If running with more than 20X[1] clients than OSTs, use a subset of clients to do I/O.
5. Create a natural partitioning of nodes so that data will go to disk in a way that makes sense.
6. Make your I/O flexible so that you can tune to the problem and machine.

5 Future Work

MPI I/O provides a mechanism whereby a programmer can supply "hints" to the MPI library. Providing hints may enable an MPI implementation to deliver increased I/O performance. There are some hints in ROMIO MPI that are recognized on the XT3/XT4 platform, including:

- `romio_cb_write`: when enabled forces parallel writes avoiding lock serialization,
- `cb_buffer_size`: defines the size of the MPI I/O buffer on each target node,
- `cb_config_list`: defines the name of the nodes participating in the I/O operation, and
- `cb_nodes`: specifies the number of target nodes to be used for collective buffering.

It is expected that some or all of these hints can help increase performance. With regard to buffering, the `cb_buffer_size` hint potentially provides a simple way for a programmer to set the buffer to an appropriate size for performance. Similarly, the `cb_nodes` hint may provide a mechanism to do the subsetting approach, which has proved that it is an effective approach when running with thousands of clients. These will be need to be tested to find out if they are effective and when.

As of the writing of this report `romio_cb_write` has been shown effective for one specific test code, while providing no benefit to another. We intend to explore the performance in additional codes.

In addition to ROMIO buffering hints, a future version of IOBUF on the Cray XT3/XT4 may add support for MPI-IO operations. We intend to track the progress of this library and do further experiments if and when it becomes available.

[1] This formula is based on the set of experiments done at ORNL. It has yet to be tested if it applies to other configurations.

Finally, it was our intention to include performance results from a real application implementing our guidelines. Due to circumstances out of our control, we were unable to include these results here. We will continue to examine real application results and, if necessary, modify our guidelines given the results.

6 Conclusions

It is clear that increasingly larger-scale supercomputers will require that application developers examine the I/O capabilities that will be available to them and determine how best to utilize them. To this end, the I/O performance characteristics of the lustre filesystem on the Cray XT3/XT4 at ORNL were investigated.

A convincing amount of data was provided that suggests several (independent) strategies an application writer should follow when designing their I/O to be run on massively parallel supercomputers. These guidelines (presented in section 4) provide fairly simple recipes for obtaining good performance in a lustre filesystem on a large Cray XT3/XT4 system. The most important suggestions are to write/read using the largest possible buffers, do parallel I/O with enough clients to saturate the filesystem, and when running at large scale (say >2K clients) use a subset of clients to perform the I/O.

7 Acknowledgments

This research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, US Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

This research used resources of the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Data for [CUSTOM1] was collected by Gene Wagenbreth.

8 About the Authors

Mark R. Fahey is a senior Scientific Application Analyst in the Center for Computational Sciences at Oak Ridge National Laboratory. He is the past CUG X1-Users SIG chair and current CUG Treasurer. Mark has a PhD in mathematics from the University of Kentucky. He can be reached at Oak Ridge National Laboratory, P.O. Box 2008

MS6008, Oak Ridge, TN 37831-6008, E-Mail: faheymr@ornl.gov.

Jeff M. Larkin is an applications engineer in the Cray Supercomputing Center of Excellence and is located at Oak Ridge National Lab. Jeff has a Master's Degree in Computer Science from the University of Tennessee. He can be reached at Oak Ridge National Laboratory, P.O. Box 2008 MS6008, Oak Ridge, TN 37831-6008, E-mail: larkin@cray.com.

9 References

1. [CFS] http://www.clusterfs.com/
2. [CUSTOM1] A custom I/O benchmark written by Gene Wagenbreth
3. [CUSTOM2] A custom I/O benchmark written by Mark Fahey
4. [IOR] IOR Benchmark, ftp://ftp.lanl.gov/pub/siop/ior
5. [LUSTRE] http://www.lustre.org/
6. [TOP500] http://top500.org/