I/O Performance on a Massively Parallel Cray XT3/XT4

Mark Fahey
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6008
faheymr@ornl.gov

Jeff Larkin
Cray Inc.
Oak Ridge, TN 37831
larkin@cray.com

Joylika Adams
Fisk University
Nashville, TN 37208
jadams49@fisk.edu

Abstract

We present an overview of the current status of input/output (I/O) on the Cray XT line of supercomputers and provide guidance to application developers and users for achieving efficient I/O. Many I/O benchmark results are presented, based on projected I/O requirements for some widely used scientific applications in the Department of Energy. Finally, we interpret and summarize these benchmark results to provide forward-looking guidance for I/O in large-scale application runs on a Cray XT3/XT4.

1 Introduction

There has been a clear trend in recent years toward increasingly larger scale supercomputers [5]. As microprocessor makers move to multicore processors to take advantage of the additional transistors on each chip, the number of processor cores in even the smallest supercomputers will become massive by today's standards. Researchers are scrambling to determine how to scale their algorithms to machines with tens and hundreds of thousands of cores, but computational performance is only one of the challenges they will face at this scale. It is critical that application developers examine their current and future I/O needs and the I/O capabilities that will be available to them so they can determine how best to use system I/O. Traditional I/O methods may no longer be sufficient at scale. The days when I/O could be treated as an afterthought to algorithmic performance have come to an end.

The Cray XT3/XT4 at the National Center for Computational Sciences (NCCS) at Oak Ridge National Laboratory (ORNL) has been on an aggressive upgrade path toward a peak of 250 teraflops (TF). Since its initial delivery, the machine has been upgraded from a peak performance of roughly 25 TF to more than 119 TF in 2007 and in excess of 250 TF in early 2008. The total number of processor cores available for application runs has increased from roughly 5,200 to more than 32,000. During this time, the I/O capabilities have also increased, as will be described later, but at a slower rate due to the high cost of I/O hardware. Users have not only had to adapt to an ever increasing number of processor cores, but also to the increasing difficulty of achieving efficient parallel I/O at scale.

To help understand the current and future state of user requirements (and in particular I/O needs), the NCCS specifically requested application requirements from the users of this machine. For the 2007 Innovative and Novel Computational Impact on Theory and Experiment (INCITE) projects with allocations on the Jaguar Cray XT3/XT4 system, the NCCS application requirements document [8] shows that the largest current data producers are the following application codes: CHIMERA, GTC, S3D, VULCAN2D, Omega3P, and POP. Most application codes were found to write restart files on a per-processor basis in an attempt to easily achieve good performance on the system. Ideally, users would write out the data via pNetCDF or parallel HDF5, thereby producing a single portable file at each restart dump. Users also consider it important to minimize the overhead of writing restart and analysis files by keeping the time to less than 5 percent of the total run time.

I/O requirements for an upcoming petaflop system were also specifically requested from users and developers of these codes. The users were asked to envision the science they would like to be doing at the time a 1-petaflop (PF) system would be available. Such use would require more than just a scaling up of their current I/O and includes scaling the science and algorithms to this new system. The results are highlighted in Table 1 on a per-simulation basis for a 1 PF leadership-class system with 200 terabytes (TB) of memory.

Table 1. Petaflop[a] application I/O requirements per simulation [8].

Code	Domain	Restart Size	Restart Freq	File Type
CHIMERA	Astrophysics	160 TB	1/hour	NetCDF or binary collective
VLUCAN2D	Astrophysics	20 GB	1/day	Binary, HDF5
POP	Climate	26 GB	1/hour	Binary, 1 serial file
S3D	Combustion	5 TB	1/hour	Binary, individual files
GTC	Fusion	20 TB	1/hour	Binary, individual files

[a] Assuming a 1 PF machine with 200 TB of memory.

Based on these I/O requirements, it is vitally important that users of these systems understand the underlying characteristics of the file system and how to take advantage of possible optimizations. Without this knowledge, simulations could waste many thousands of hours of their allocations doing I/O. For example, if we consider the combustion code S3D, which needs to output 5 TB of data every half hour, and if we assume the file system and I/O routines could sustain 10 GB/s of throughput during the simulation (most codes rarely achieve this level of I/O bandwidth, so the assumption is overly optimistic in favor of the file system), it would take 500 seconds (nearly 8.5 minutes) to write out the restart file. This approach would mean I/O accounts for approximately 15 percent of the wall time between restarts, which is much higher than the 5 percent threshold required by users.

To this end, it is our goal to present the methodology we used to gather and interpret basic I/O performance characteristics on the ORNL XT3/XT4. This methodology is sufficiently general to be used on any machine, particularly Cray XTs and similar platforms with high bandwidth file systems.

In Section 2 we provide background information on the ORNL XT3/XT4 and its Lustre file system as well as a brief MPI-IO overview as it pertains to our benchmarks. In Section 3 we then present a large amount of I/O performance data for many scenarios and interpret the results. Most of the data are from MPI-IO benchmarks, but some HDF5 benchmark data are also included. We present our conclusions in Section 4.

2 Background

This section provides an overview of the ORNL Cray XT3/XT4 and its associated Lustre file system. It also provides a brief summary of MPI-IO as it pertains to the benchmarks used in this study.

2.1 Cray XT3/XT4 at ORNL

The work described in this paper was performed on the Cray XT3/XT4 at the NCCS at ORNL. The XT3 and XT4 represent a line of massively parallel processor products from Cray, with the XT4 being an evolutionary descendent of the XT3. These systems are designed around the AMD Opteron processor, a scalable custom interconnect (Cray SeaStar), a lightweight kernel operating system (Catamount), and a Lustre file system from Sun Microsystems [4]. See references [6, 11] for more details on the Cray XT3 and XT4 architectures and in particular the XT3/XT4 system at ORNL.

It is important to note that as part of ORNL's path to constructing a petaflop computer at the NCCS, this system went through several software upgrades in a very short period of time during the course of this study. The operating system (OS) was upgraded several times a month, and the default compiler was upgraded often as well. Although both OS and compiler changes can affect I/O performance, the primary factor is the configuration of the file systems, not the software versions. We found that although compiler and OS versions vary between some of the benchmark results presented below, the effect of these changes is small and does not impact our conclusions.

Please also note that ORNL's Cray XT3/XT4 underwent major changes after this paper was submitted. The OS was swapped from Catamount to Compute Node Linux (CNL), and all the nodes were upgraded yet again from dual-core to quad-core, with a corresponding clock rate decrease from 2.4 GHz to 2.0 GHz. This made it unfeasible to gather additional data with any sort of consistency.

2.2 Lustre Basics

High bandwidth I/O on Cray XT3/XT4 systems is provided by the Sun Microsystems Lustre file system [3]. Lustre [2] is an object-based parallel file system designed to provide large, high-bandwidth storage on large, clustered computers. A Lustre file system has one or more Object Storage Servers (OSSs), which handle interfacing between the client and the physical storage. Each OSS serves one or more Object Storage Targets (OSTs), where the file objects are stored to disk. The term "file striping" refers to the number of OSTs used to store a file. For example, if a file has a stripe count of four, then it is broken into objects and stored on four OSTs in round-robin fashion. The file system namespace is served by a Metadata Server (MDS). As the name implies, the MDS is a database that holds the file metadata for the entire file system. Whenever a metadata operation occurs, such as an open or file creation, the client must poll the MDS. At this time, a Lustre file system is limited to one MDS, which can result in a parallel I/O performance bottleneck at large scale. Figure 1 depicts how the Lustre architecture is connected to compute nodes of a Cray XT3/XT4 system.

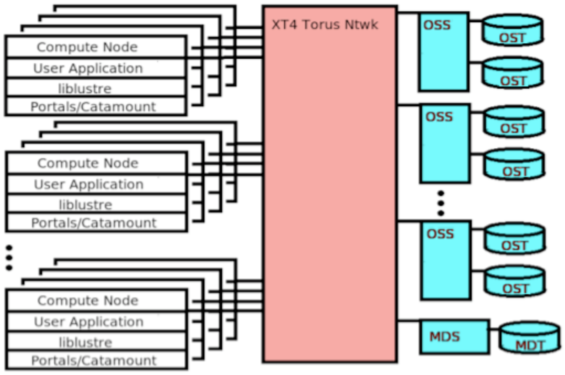


Figure 1. Lustre architecture.

Files are striped over one or more OSTs when they are written to the file system. Users have 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. However, a file cannot have a stripe width greater than the total number of OSTs configured in the host file system.

2.3 Lustre on Jaguar

During the time the authors were collecting data for this paper, the Lustre file system on the Cray XT3/XT4 system at ORNL underwent several configuration changes, which are summarized in Table 2. While a having consistent configuration for these studies would be desirable, the aggressive upgrade path at the NCCS has made this difficult. Valid trends can still be observed by evaluating the data collected over the course of the system upgrades. To that end, the file system configurations available during the study are as follows: The first configuration (LC1) had a total of 96 OSTs over 48 OSSs. These were configured over 12 DataDirect Networks (DDN) 8500 controllers (also known as couplets) with 8 OSTs per couplet connected to the storage backend over 2 Gb/s fibre channel. The disk controllers were configured with two tiers per logical unit number, where a tier is a DDN term for a nine-disk, 8+1 RAID array. The theoretical peak performance of this file system was 2 GB/s × 12 couplets = 24 GB/s.

During a short testing period, the file system was configured (LC2) to 160 OSTs spread over 80 OSS nodes connected to DDN 9550s via two 4 Gb/s fibre channel cards per OSS. The theoretical peak of this configuration was 80 GB/s. This file system was then reconfigured into incarnation (LC3), consisting of 144 OSTs over 72 OSSs connected to DDN 9550s via two 4 Gb/s fibre channel cards per OSS. The theoretical peak of this configuration was 72 GB/s.

It should be noted that the Lustre I/O performance is strongly dependent on the file system configuration and weakly dependent on the system architecture when the nodes performing I/O cannot saturate the file system. Differences in performance between the file system configurations listed are due to the change in disk controllers.

2.4 MPI-IO Basics

A standard interface for parallel I/O was added to the Message Passing Interface (MPI) specification in the MPI2 specification [7]. Cray provides MPI-IO functionality via the ROMIO package [10] developed at Argonne National Laboratory. The specification provides several methods for performing parallel I/O, which can be a cause for confusion for users. Two methods specifically used in this paper were explicit file offsets and collective I/O with fileviews. MPI-IO provides several methods for performing I/O to a shared file at explicit offsets (e.g., `MPI_FILE_WRITE_AT` and `MPI_FILE_READ_AT`), requiring the user to calculate a file offset for each process participating in an I/O operation. This approach may be convenient when an application has a very regular I/O pattern.

Table 2. ORNL Cray XT3/XT4 Lustre configurations.

Configuration	Machine	Disk Controllers	# OSSs	# OSTs	Theoretical Peak
LC1	XT3	12 DDN-8500 couplets	48	96	24 GB/s
LC2	XT4	20 DDN-9550 couplets	80	160	80 GB/s
LC3	XT3/XT4	18 DDN-9550 couplets	72	144	72 GB/s

Additionally, users may perform collective I/O operations, where all participating processes call an I/O function and allow the library to handle file partitioning, data transfer, buffering, etc. Users can establish a file-view when opening a file for collective I/O operations, which guides the library on how to partition the file so that a process "sees" only the part of the file that it needs without having to worry about calculating a file offset.

MPI-IO also 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. This mechanism is discussed in Section 3.2.6, which compares MPI-IO and HDF5 performance.

This is far from an exhaustive explanation of MPI-IO methods, but it should provide sufficient information for an understanding of results presented below.

3 Benchmark Methodology and Results

In this section, we present I/O performance data obtained from three different benchmark codes. I/O performance is studied independently of applications because there is no standard method for carrying out I/O across codes or even within a code. In fact, application benchmarks with I/O enabled can often overemphasize the I/O requirements. Instead, here we attempt to understand some basic characteristics of the XT3/XT4 file system with relatively simple benchmarks. Additional I/O benchmarks and application results can be found in [12].

Below we describe three codes that were used to test various aspects of I/O performance of the Lustre file system detailed in Section 2. Because users rarely have the luxury of running with a dedicated system, the authors collected data during normal operation of the system without dedicated access, intending to produce results that would be representative of actual application runs.

3.1 Benchmark Codes

3.1.1 Code 1

Code 1 is a custom code (written by Gene Wagenbreth) designed to emulate writing a large amount of data to disk from all processors. This code is a very simple, buffered, single-file MPI-IO write benchmark that was designed to model the I/O pattern of a specific application. The benchmark was run only to processor counts appropriate to the application it was modeling. The benchmark models the behavior of efficiently exporting a large amount of data from memory to disk, as would be done in checkpointing. The benchmark version we used opens a file across all tasks, but in task counts comparable to the expected number of writers for large program runs. Each writer in the benchmark uses the `MPI_FILE_WRITE_AT` method to write data to a calculated offset in the shared file. This approach, of course, assumes a regular amount of data is written by each processor, which makes the calculation of offsets and distribution of I/O operations trivial. Furthermore, the application also assumes that the time to transfer data to the writers is trivial in comparison to time required for I/O.

3.1.2 Code 2: IOR

IOR (Interleaved Or Random) [1] is a parallel file system test code developed by the Scalable I/O Project at Lawrence Livermore National Laboratory. This 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.

3.1.3 Code 3

Based on results gathered from the above benchmarks, in later sections we will present results from which the idea of using a subset of MPI tasks to do the I/O for an entire application was investigated. Thus, another custom Fortran code (written by Mark Fahey of ORNL) was designed to write out contiguous buffers by each process in a subset of MPI_COMM_WORLD either to a single-shared file or, alternatively, to one file per process. Furthermore, this code was designed to accept run-time parameters defining a subset of processes with which to do I/O and to use one of several methods of moving the data to the subset of processes that do the I/O. (The author of Code 3 is not aware of this functionality being available in any standard I/O package.) The intent (as with the code described in Section 3.1.1) was to model the behavior of writing a large amount of data to disk, as in a checkpoint. This technique of using a subset of processes to perform I/O is referred to as aggregation.

3.2 Benchmark Tests

All results reported below were generated on the XT3/XT4 machine running Unicos/lc 1.5. For all tests, the Lustre file system stripe sizes and stripe counts were set using the `lfs setstripe` command prior to running the benchmarks. The stripe count was set to the maximum, and the stripe size was set to one megabyte (MB) for all tests unless otherwise noted.

As discussed above, results were obtained with different configurations of the file system described in Table 2. Note that all results were obtained in nondedicated mode; that is, the machine (and the Lustre file system) were being used by multiple users, as would be typical for actual application runs.

3.2.1 Single Writer Performance

First I/O performance on a single task was studied. Because it unfortunately is still common practice to send all data to a single node to be written to disk, it is worthwhile to investigate the efficacy of this practice. This method suffers from networking overhead when transferring data to the single writer and a bottleneck when outputting all a program's data through a single node. Writing from a single node simply will not saturate the file system bandwidth needed for large I/O operations, but it can give some insight for tuning parallel I/O operations.

Benchmark Code 1 was used to perform these tests. It was compiled with version 6.2 of the Portland Group (PGI) compiler suite. The Lustre file system was set up as configuration LC2 for the tests in this section.

Figure 2a illustrates the bandwidth of a single writer to a single OST, varying the user-allocated buffer and

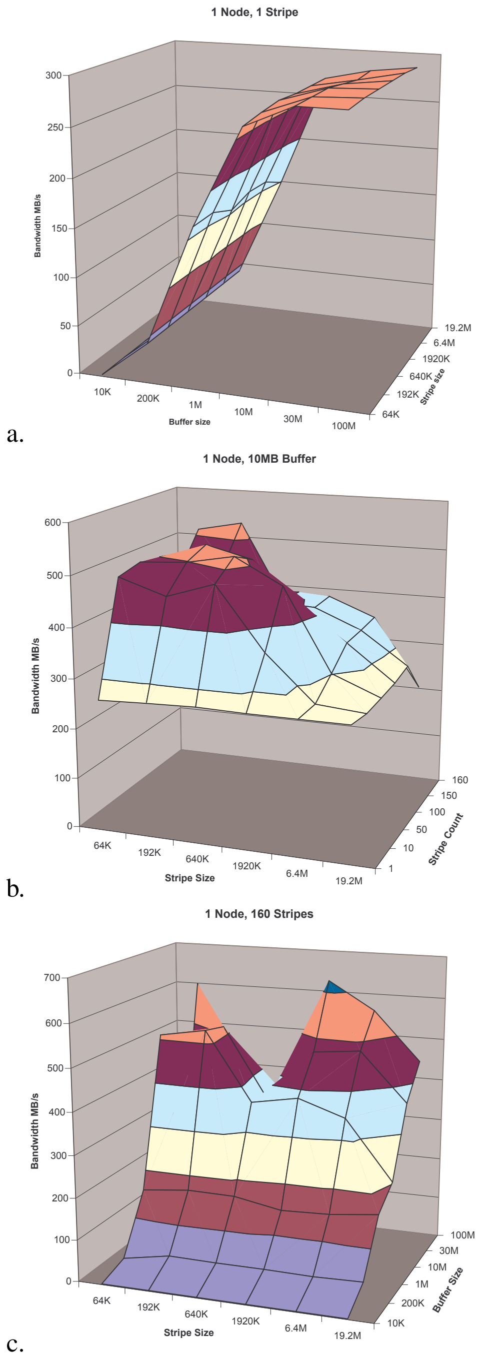


Figure 2. Bandwidth when writing from one client (a) to one stripe, varying the size of the buffer and stripe; (b) with a 100 MB buffer, varying the stripe size and width; and (c) to a file with a stripe width of 160, varying the size of the buffer and stripes.

stripe size. It is clear that one writer and one OST will not achieve high bandwidth due to the limited bandwidth available to a single OST (roughly 500 MB/s for this configuration), but the striking observation gleaned from this figure is the importance of buffering I/O operations. Varying the size of the file system stripe had little effect on the bandwidth, but varying the size of the user buffer greatly affected the bandwidth. The need to buffer I/O operations should be obvious, and Figure 2a shows that a 1 to 10 MB buffer can significantly improve write performance.

Figures 2b–c show further evidence of the need to use multiple writers to achieve reasonable bandwidth. No matter how widely the file was striped, this benchmark was unable to achieve greater that 600 MB/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 file system bandwidth.

Collectively, the data indicate that when using one I/O task, a code cannot take advantage of a large Lustre file system, as would be expected. Furthermore, stripe counts greater than one do have a positive effect, but only up to a relatively small amount, around 10 to 20 percent. On the other hand, stripe size is essentially irrelevant in this scenario, except when the stripe size is set too large.

3.2.2 Fixed Number of Stripes

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

Code 1 was compiled with version 6.2 of the PGI compiler suite. The Lustre file system was set up as configuration LC2 for the tests in this section.

The graphs in Figures 3a–c plot the I/O performance for a fixed stripe count of 150, while varying the stripe and buffer sizes along the axes and the number of writers between graphs. Figure 3 shows 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 write bandwidth. The difference between the bandwidth at 100 and 150 writers is negligible.

The results indicate that not only may I/O rates be improved through the obvious approaches of using a buffer size of at least 1 to 10 MB and striping across a large portion of the available OSTs, but also that a sufficient

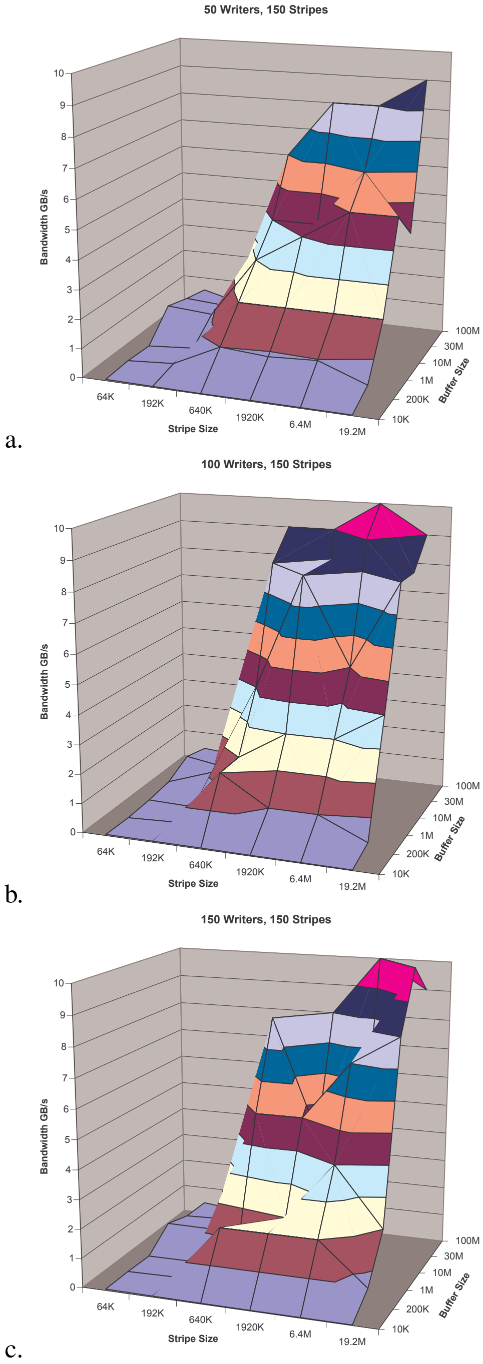


Figure 3. Write bandwidth for 150 stripes with (a) 50, (b) 100, and (c) 150 writers. The bottom and side axes of each graph are the stripe size and buffer size, respectively.

number of I/O nodes must be used to sustain high performance.

These graphs suggest that there is no reason to have less than a one-to-one relationship between writers and stripes. This particular benchmark was not used to explore using more writers than OSTs; see Sections 3.2.4 and 3.2.5.

3.2.3 Fixed Number of Writers

Similarly, fixing the number of writers and varying the other parameters can provide additional insights. Benchmark Code 1 was compiled with version 6.2 of the PGI compiler suite. The Lustre file system was set up as configuration LC2 for the tests in this section.

The graphs in Figures 4a–c illustrate the need for buffering by varying the size of buffers between graphs. The significant improvement in performance as the buffer size increases should be noted. Buffer sizes below 1 MB are not shown, but they result in poorer performance. While many applications are unable to sacrifice 10 to 100 MB for use in I/O buffers, these graphs make it clear that whatever sacrifice can be made for I/O buffers will result in improved I/O performance.

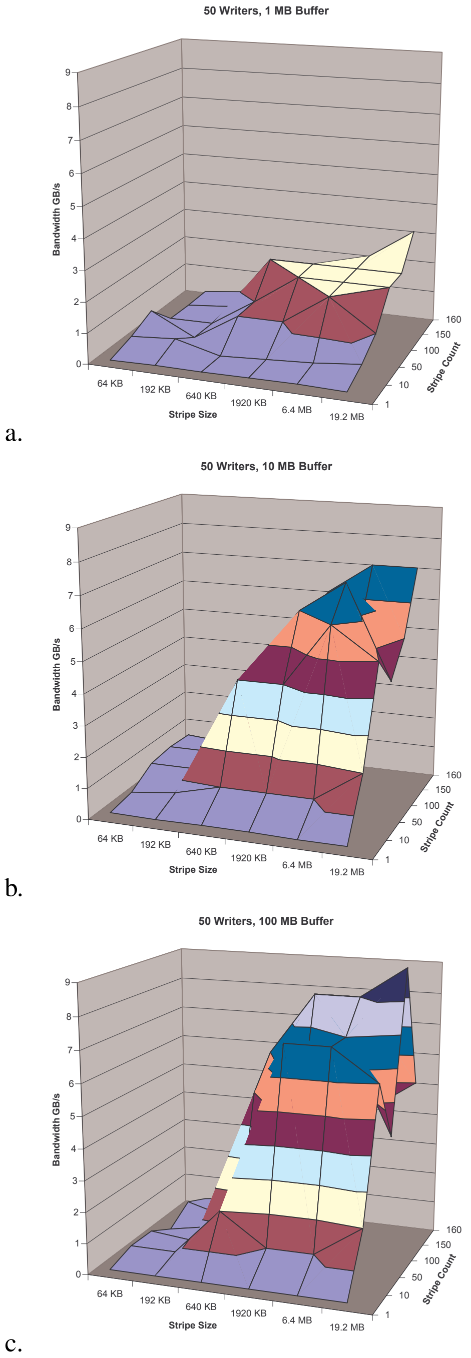
The clearest result from the above benchmarks is the importance of buffering I/O operations. While adjusting the stripe size and width does provide noticeable gains in write performance, the use of large I/O buffers seems to have the most pronounced effect on performance. This benchmark also substantiates the idea of using at least as many writers as the stripe count.

3.2.4 Scaling Results

The benchmarks thus far have tested the file system with up to 150 clients. A major thrust of this research was to test the Lustre file system at scale; that is, to test the file system doing parallel I/O out to many thousands of clients. Some scalability issues are seen only when testing with a large fraction of the available resources.

Benchmark Code 2 was used to test the scalability of the Lustre file system by performing parallel I/O tests out to many thousands of processors. Code 2 was compiled with version 6.1 of the PGI compiler suite. The Lustre file system was set up as configuration LC3 for the results in this section.

Figure 5 shows the performance results when using IOR with constant buffer size per client (core) and increasing the number of clients. The top plot in Figure 5 is the case when writing or reading with one file per client, while the bottom graph is for a shared file. The maximum achieved bandwidths are 42 GB/s (read) and



a.

b.

c.

Figure 4. Write bandwidth for 50 writers with (a) 1 MB, (b) 10 MB, and (c) 100 MB buffers. The bottom and side axes are stripe size and stripe count, respectively.

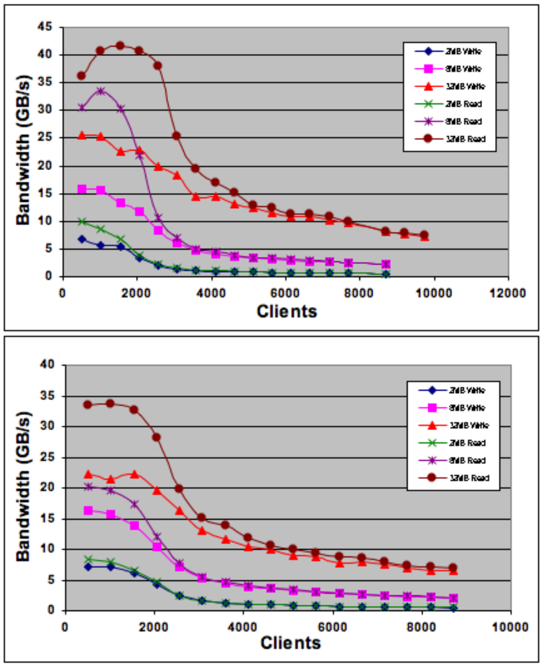


Figure 5. These graphs fix the buffer size per core at 2, 8, or 32 MB for writes and reads and vary the number of clients along the x-axis. Output was to (top) one file per process and (bottom) a single shared file.

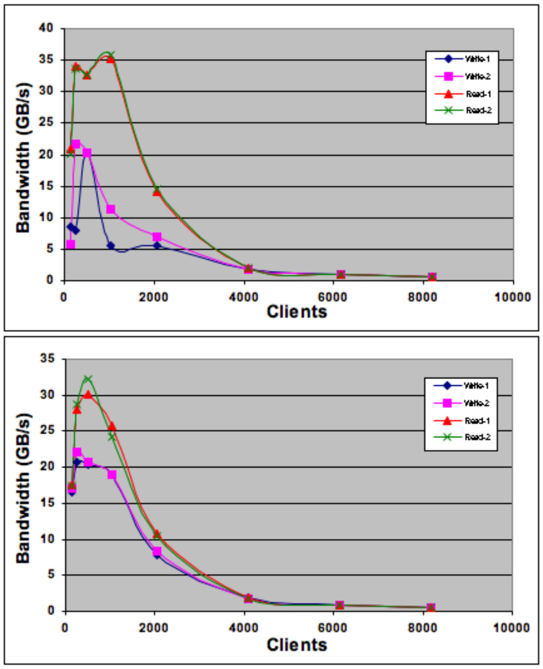


Figure 6. These graphs fix the aggregate file size at 16 GB and vary the number of clients along the x-axis for writes and reads. Output was to (top) one file per process and (bottom) a single shared file. The "-1" and "-2" in the legend represent two separate sets of runs.

25 GB/s (write) for one file per client and 34 GB/s (read) and 22 GB/s (write) for a shared file.

The scalability of Lustre was also tested by keeping the aggregate file size constant while increasing the number of clients in 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 6 and 7, the aggregate sizes of the files were 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 6 and 7, the top plots show the performance for one file per client, while the bottom depicts the shared-file performance.

In Figure 6, 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 one file per process and 33 GB/s for a shared file, very similar to the results shown above. Similarly, in Figure 7 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 for 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.

The results above clearly illustrate that the achievable bandwidth drops dramatically in all three scenarios (Figures 5–7) for both one file per process and a shared file after it reaches a maximum somewhere around 500 to 2,000 clients. We are not certain why the performance eventually decreases to an asymptotic level—the same behavior occurs for both read and write—but conjecture the following: Much of this decrease can be attributed to an overloading of the single MDS, as mentioned in Section 2.1. Reference [12] presents data showing a significant increase in the time required for an open as the number of clients increases. We additionally speculate that the increased number of clients per OST causes additional overhead, which reduces the achievable bandwidth per OST.

Regardless, it seems clear that using a subset of the

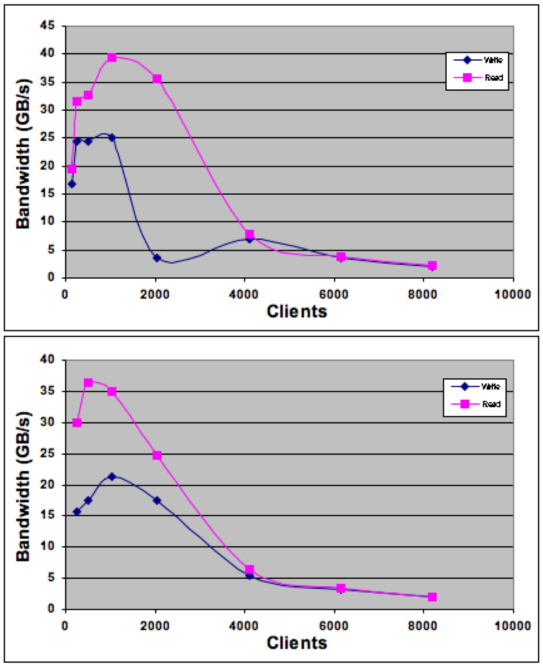


Figure 7. These graphs fix the aggregate file size at 64 GB and vary the number of clients along the x-axis for writes and reads. Output was to (top) one file per process and (bottom) a single shared file.

application tasks to do I/O will result in better overall performance at scale. Additionally, these plots indicate that using more than one client per OST is the only way to get the practical peak performance because the maxima occur between 500 and 2,000 clients, and the file system has 144 OSTs. And a third observation is that the performance when doing I/O to a shared file and to one file per task are qualitatively different, but is it unlikely that the difference is sufficient to change how users run on the file system.

3.2.5 Subsetting Results

The results in the previous subsection suggest that using a subset of nodes for I/O may provide better bandwidth than using all the available processes to do I/O. Code 3 was used to investigate this hypothesis and was compiled with version 7.0 of the PGI compiler and run on Lustre configuration LC3.

Table 3 summarizes the results of running Code 3 with 12,288 processes in which each process had an 8 MB buffer that needed to be checkpointed. All these runs performed I/O to a single shared file. As with most of the results presented here, the runs were in nondedicated mode, and we report the maximum over three runs.

As alluded to in Section 3.1.3, the code has multiple methods of doing parallel I/O. These are as follows:

mpiio This method involves straightforward use of MPI-IO routines in which every process takes part in I/O.

agg For each subset, all the processes in a subset send their data to the group's master process, which then writes out one large buffer.

ser For each subset, each process sends its data to the master process in order, and when received the data are immediately written out. In effect, the data are written out in parallel across the subsets, but serially within each subset.

swp For each subset, each process writes out its own data in order according to its rank within the subset. This can be thought of as sweeping through the subsets. I/O is serial within a subset, but each process writes out its own data in parallel across the subsets.

Note that all methods use MPI-IO routines to actually perform I/O. The *mpiio* and *swp* methods involve all processes in I/O, but *agg* and *ser* involve only some subset of processes. The *agg* method has been implemented with both collective (`MPI_FILE_SET_VIEW + MPI_FILE_WRITE`) and independent (`MPI_FILE_WRITE_AT`) MPI-IO calls. The *mpiio* method is the only other collective method we have implemented to date. The other methods (*ser* and *swp*) use independent MPI-IO functions.

In Table 3, the 1,536 entry indicates that there were 1,536 subsets of eight processes each. Similarly, for the 768 entry, there were 768 subsets of 16 processes each, and for 512, there were 512 subsets of 24 processes each.

The *mpiio* method shows that straightforward use of MPI-IO with 12,288 processes results in 5.4 GB/s I/O throughput. The *agg* and *ser* methods demonstrate that the subsetting methods can achieve much higher bandwidths, in particular as high as 29 GB/s for the case with 1,536 subsets. These tests provide early evidence that using a subset of tasks for I/O produces significantly better performance than using all the available cores at scale.

Although three different subset sizes were tested in what appears to be the optimal range of I/O writers, it is not clear from the results if there is an optimal subset size or even an optimal I/O method for a particular subset size.

Table 3. MPI-IO write bandwidth in GB/s.

	MPI-IO Number of Subsets and Method									
	12,288	1536			768			512		
	mpiio	*agg*	*ser*	*swp*	*agg*	*ser*	*swp*	*agg*	*ser*	*swp*
Collective	5.4	19.6			21.4			19.1		
Independent		29.0	20.9	5.4	27.4	25.1	5.4	25.0	25.4	5.6

3.2.6 MPI-IO and HDF5

In this section, results comparing the performance of MPI-IO and HDF5 using Code 2 are presented. As mentioned in Section 1, many users wish to use a portable I/O library like HDF5. Therefore, understanding the performance of HDF5 (especially relative to MPI-IO) is essential.

Code 2 was compiled with version 7.0 of the PGI compiler suite. The Lustre file system was set up as configuration LC3 for the results in this section.

Figures 8 and 9 show results when running IOR to test both the MPI-IO and HDF5 interfaces. The IOR blocksize (`-b`) was set to 16 MB. A trend line is plotted for various transfer sizes indicated by `-t`. Also, `-c` indicates that IOR used a collective call, and `-V` indicates MPI-IO fileviews were used. Lastly, `hint` indicates an MPI-IO hint was specified, and `EnvVar` indicates a Lustre-specific environment variable was set.

The write performance data from Figure 8 show that HDF5 performance is nearly the same as that of MPI-IO. The plots do show that one must be careful when using collective HDF5 or fileviews with MPI-IO (at least within the context of IOR) where performance will be tens of megabytes per second rather than gigabytes per second. The read performance data from Figure 9 show a much different picture than for writes. HDF5 performance reaches an asymptotic bandwidth around 1.6 GB/s, with some cases reaching approximately 2 GB/s at 512 clients. On the other hand, MPI-IO performance attains a bandwidth of 18 GB/s when the transfer size is large (16 MB) and a bandwidth of 13 GB/s for the smaller transfer size of 256 KB when doing collective I/O. Basically, there is an order of magnitude difference in HDF5 performance as compared to that of MPI-IO.

Note that for some of these runs, some lesser-known optimizations were employed to get the best performance. For example, the environment variable `MPICH_ROMIO_RECORD_LOCKING` was set to 1 when performing collective I/O with HDF5 to raise 10 MB/s performance to gigabytes per second. And when using the fileview option with the MPI-IO method in IOR, it is essential that the MPI-IO hint `romio_ds_[read,write]` be set to "disable" or performance will be in the low megabytes per second as well. (The "ds" stands for data sieving [9].)

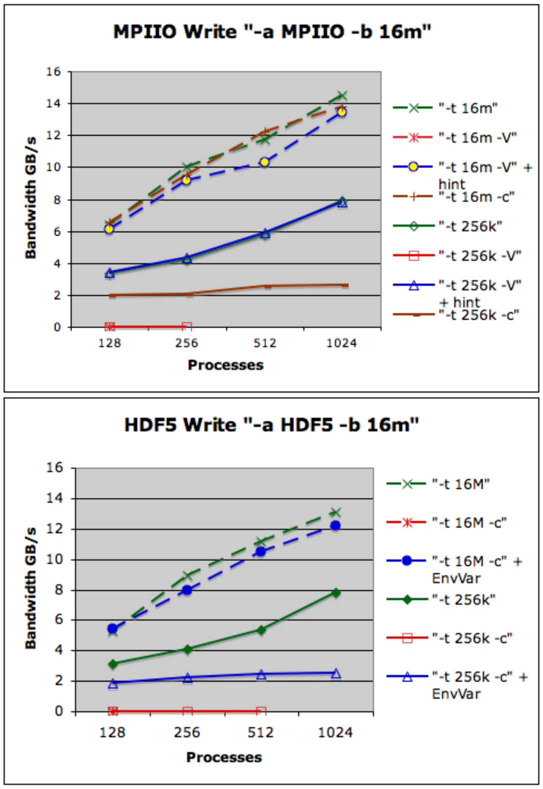


Figure 8. MPI-IO (top) and HDF5 (bottom) write performance.

4 Conclusions

It is clear that application developers will face many new challenges as computing systems build up to and beyond the petaflop level, not the least of these will be I/O. I/O subsystems are simply unable to keep up with system growth due to cost and technological limitations. At the NCCS at ORNL, scientists are working to understand both current and future I/O needs of users and

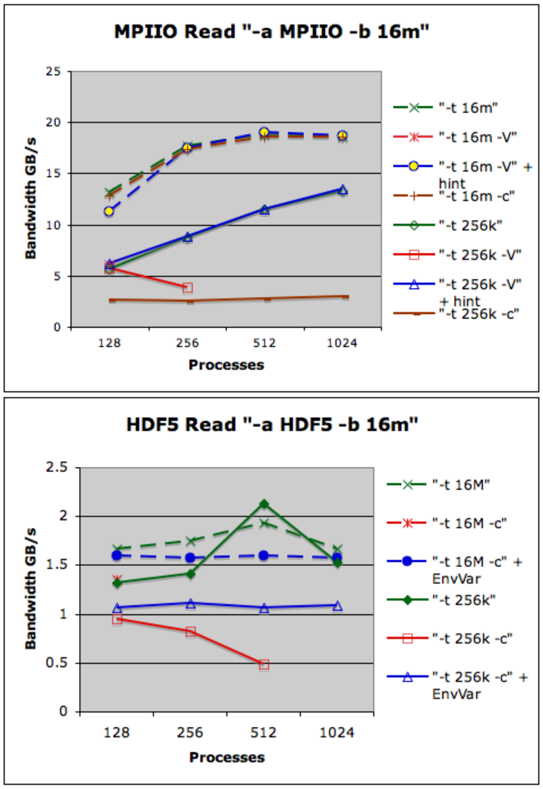


Figure 9. MPI-IO (top) and HDF5 (bottom) read performance.

to use this information to make decisions about future systems. It is also critical for users to understand their application's current and future I/O needs, taking into consideration the increasing complexity of the science problems that will be solved on these larger machines. Users must also be aware of the I/O capabilities of the systems they use and how to best exploit the system's strengths. Admittedly, this is a very difficult task, so we make the following observations based on our benchmark data.

The data from Code 1 underscore the importance of buffering I/O into large operations. While a large I/O buffer does decrease the memory available for computation, having a 10 to 100 MB buffer can significantly improve I/O performance and lessen the impact of I/O on application run time. These data also show that using more OSTs than I/O processes in a Lustre file system simply does not make sense. Code 2 demonstrates the importance of performing very large scale I/O over a smaller subset of available processes. It was observed that on the particular file system benchmarked with 144 OSTs, using 500 to 2,000 I/O processes seemed to give the best I/O performance. Subsetting has the added benefit of improving I/O buffering as well. Code 2 also showed that given the proper configuration, parallel HDF5 writes can achieve performance comparable to that when using MPI-IO directly; however, HDF5 read performance was an order of magnitude worse than for MPI-IO for our tests. Code 3 tested several methods of doing I/O subsetting and confirmed that I/O subsetting substantially improves I/O performance for very large processor counts.

Some straightforward I/O performance characteristics on the ORNL XT3/XT4 have been presented. Benchmarks designed to represent applications with demanding requirements in mind were performed; the lessons learned are of clear benefit to users of Cray XTs and similar platforms with high bandwidth file systems. Because the data presented in this paper were collected with specific I/O configurations, some of our conclusions may apply only to similar configurations.

It is unlikely that a set of firm rules can be developed for I/O operations, so as with Code 3, application I/O should be written in a manner that is flexible enough to be adapted to system capabilities. Upon porting a code or problem to a specific machine, users may then model their application's I/O and quickly determine a reasonable set of I/O parameters to use. Just as algorithmic kernels can be used to easily and quickly tune an application's performance on a given system or architecture, modeling an application's I/O can help improve overall application performance and productivity.

Acknowledgments

This research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy under contract number DE-AC05-00OR22725 with UT-Battelle, LLC. This research was also sponsored by the Research Alliance in Math and Science program, which is sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy.

This research used the resources of the NCCS at ORNL, which is supported by the Office of Science of the U.S. Department of Energy under contract number DE-AC05-00OR22725.

Data for Sections 3.2.1 through 3.2.3 were collected by Gene Wagenbreth, who wrote Code 1.

References

[1] IOR benchmark. ftp://ftp.llnl.gov/pub/siop/ior.

[2] Lustre. http://www.lustre.org/.

[3] Sun Luster File System. http://www.sun.com/software/products/lustre.

[4] Sun Microsystems. http://www.sun.com/.

[5] TOP 500. http://top500.org/.

[6] S. R. Alam, R. F. Barrett, M. R. Fahey, J. A. Kuehn, J. M. Larkin, R. Sankaran, and P. H. Worley. An early evaluation for petascale scientific simulation. In *Proceedings of the ACM/IEEE SC2007 Conference*, 2007.

[7] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI The Complete Reference: Volume 2 - The MPI Extensions*. MIT Press, September 1998.

[8] D. Kothe and R. Kendall. Computational science requirements for leadership computing. Technical Report ORNL/TM-2007/44, ORNL, July 2007.

[9] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[10] R. Thakur, E. Lusk, and W. Gropp. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. Technical Report Technical Memorandum ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 2004.

[11] J. S. Vetter, S. R. Alam, T. Dunigan, M. R. Fahey, P. Roth, and P. H. Worley. Early evaluation of the Cray XT3. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2006.

[12] W. Yu, J. Vetter, and H. S. Oral. Performance characterization and optimization of parallel I/O on the Cray XT. In *Proceedings of the 2008 IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, 2008.