Petascale IO Using The Adaptable IO System

Jay Lofstead (lofstead@cc.gatech.edu)
Scott Klasky (klasky@ornl.gov)
Michael Booth (Michael.Booth@Sun.COM)
Hasan Abbasi (habbasi@cc.gatech.edu)
Fang Zheng (fang.zheng@gatech.edu)
Matthew Wolf (mwolf@cc.gatech.edu)
Karsten Schwan(schwan@cc.gatech.edu)

1 May 2009

Abstract

ADIOS, the adaptable IO system, has demonstrated excellent scalability to 29,000 cores. With the introduction of the XT5 upgrades to Jaguar, new optimizations are required to successfully reach 140,000+ cores. This paper explains the techniques employed and shows the performance levels attained.

1 Introduction

Many new systems today already exceed 100,000 cores. This order of magnitude expansion of compute capacity has not always been matched by IO capacity. While the number of IO nodes may proportionally match the compute expansion, the impact on metadata services and the parallel capacity of the storage system can be lacking. For example, the Lustre [2] scratch system on Jaguarpf at Oak Ridge National Laboratory has 672 storage targets, but internally Lustre is limited to 160 storage targets for a single file. The amount of concurrent access to the storage target can also greatly limit IO performance. With a 100,000 core job running against 160 storage targets, on average, 625 processes will write to each storage target simultaneously. This assumes that the user has bothered to configure the output directory to use the maximum stripe count. Otherwise, if it defaults to something small like 4, as many as 25,000 or more process may attempt to write to a single storage target at the same time. Achieving the advertised IO rates for these large systems requires considering these facts and managing how IO is performed. ADIOS has eliminated the need for the user to change application code when selecting different IO methods with great success [7]. As codes are tested on these new machines, different techniques for IO acknowledging the system characteristics are required.

Two approaches are possible for addressing these concerns. First, asynchronous data movement to a 'staging area' can offload the time spent performing IO to during non-communication phases of the computation thereby taking advantage of a relatively quiet network. In addition to simple data movement, 'data services' can be performed during the data movement minimally slowing the data extraction and writing operation. The savings in compute time can be enormous with a small investment of additional compute resources for the staging nodes. The additional potential cost is continuing execution of the code even though the IO failed thereby wasting compute time. While the risk is not large compared with the gain in performance, it is unacceptable in some circumstances. For these cases, highly optimized synchronous data movement can be employed.

By carefully managing the characteristics of the data, the concurrency required during IO, and how the underlying parallel file system is configured, high performance synchronous IO can be achieved. Controlling metadata server impact by serializing 'open' calls [9] has proven valuable previously. In this work, controlling access to storage targets and properly distributing the IO workload has demonstrated a peak of 83 GB/sec out of a potential of 110 GB/sec.

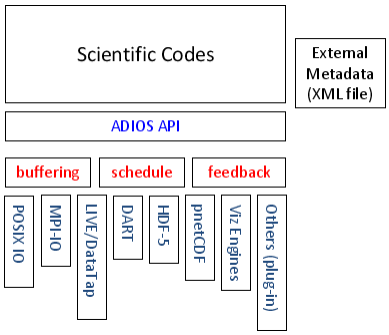

Figure 1: ADIOS Architecture

The rest of this paper will discuss related work, an overview of the architecture, discuss the evaluation system and setup, provide some conclusions, and finally some discussion of future work.

2 Related Work

Using staging areas to accelerate IO performance has been explored recently [11, 3]. While these approaches work well, they all require some additional resources. Depending on the size of the data, the additional resource cost in space may outstrip the benefits. Approaches like our DataTap [4, 1] services addresses these concerns by adopting asynchronous data transport while carefully managing the impact on communication within the application code.

Many systems have chosen to avoid this issue entirely and hope that a lower level layer will address the issue. Parallel HDF-5 [5] and Parallel NetCDF [10] build on top of MPI-IO and rely on that layer to properly manage IO. While it has been successful in the past, now that the storage system capacity is exceeding the per-file maximum sizes, different techniques are critical to achieving high IO performance.

Our own ADIOS [7, 8, 9] work has demonstrated the importance of a configurable IO layer and changing the IO technique based on platform, data sizes, and size of the job. While these optimization approaches work well, by managing the IO more aggressively in concert with the system settings, much better performance can be achieved.

3 Architecture

The ADIOS architecture 1 is unchanged from our previous work. The only addition is a new transport method created for the new scheduled IO technique. Two different approaches are considered for this work.

3.1 Scheduled IO

By scheduling when each process can write to the storage target, less concurrency-related inteference will be induced. This is achieved through three techniques:

- Split files - by splitting the output into more files, the system is better able to use all of the available storage targets. While this does impose an additional overhead of managing more files, the code does not see a difference from working with a single output file.
- Scheduled metadata operations - rather than bombarding the metadata server with all processes simultaneously to open the output file(s), the system explicitly serializes the 'open' calls for each file resulting in faster overall open times. Relying on the metadata server to manage the serialization results in overloading of the metadata server and all processes receiving worse service.
- Schedule IO - by reducing the concurrency to the rotational media in the storage targets, performance more closely related to streaming can be achieved.

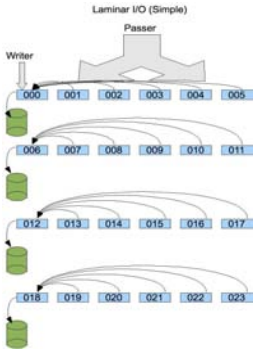

Figure 2: Simple Aggregation

The base method shown in the figures employs an independent MPI-IO approach using scheduled metadata operations with a few other adjustments. First, the stripe size for each file created is set to be the rounded up size of the largest data written for any process writing to that file. Second, the write offset for each process is set to stripe boundaries. This does introduce some empty space in the file, but it reduces false sharing and OST spanning of any write operation.

3.2 Aggregation

A second configuration consists of using aggregation through a single node to leverage repeated use of the same network paths. This approach uses a different file split algorithm and currently does not schedule the metadata operations. Specifically, this approach splits the processes into per-file groups, like the scheduled IO transport, but writes all of the data for each file from only one process assigned to that file. The unique characteristic of this aggregation approach is that no additional memory is required for the aggregation. Current MPI-IO aggregation methods generally require the aggregation node have enough memory to collect all the buffers from the other processes before writing. Using this setup, two different approaches for aggregation are considered.

3.2.1 Simple Aggregation

For simple aggregation illustrated in figure 2, all of the processes for a file send data to a master process that writes the data.

For this approach, more of the network is likely traversed between any node and the master writer. This may cause network inteference that could slow performance.

3.2.2 Brigade Aggregation

For brigade aggregation, illustrated in figure 3, each process writes to its predecessor marching the data towards the master that writes the data.

This avoids this potential interference by localizing all of the data transfers at a cost of moving the data potentially multiple times before it is written.

4 Evaluation

The evaluation is performed on the petaflop partition of the Jaguar machine at Oak Ridge National Laboratory. It is a Cray XT5 machine with 18,680 nodes each with dual, quad-core AMD Opteron processors (149,376 cores) and 2 GB of RAM per core. The scratch filesystem is a 672 storage target Lustre 1.6 system with 10 PB of storage.

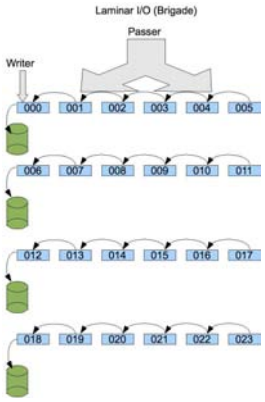

Figure 3: Brigade Aggregation

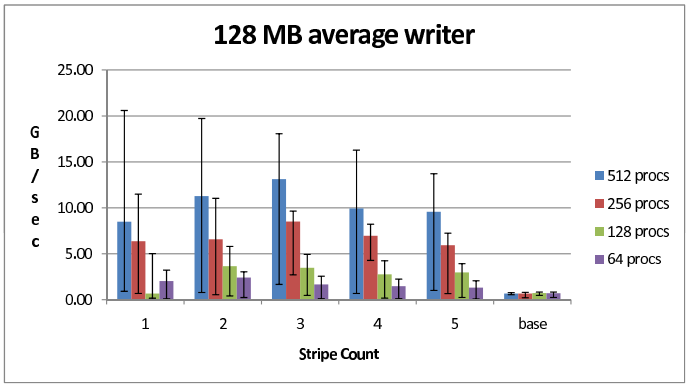

Figure 4: 128 MB per Writer

4.1 Staging Performance

This evaluation is performed on a variety of core counts from 64 to 65536 with a file stripe count of 1 through 5. As a base comparison, a system default file is created for most of the process counts. This has a stripe count of 4 used for a single file across all processes.

The evaluation application is a simple ADIOS code carefully configured to avoid local node caching. All data size measurements only include the raw application data. The time measurements include the file open, writes, index collection, and close operations. With all of these operations included as part of the time measurement, the raw write performance is underrepresented in these results. More detailed timing results are in development. All combinations except for the larger stripe counts at 64K cores are run at least 3 times. All of the small scale results are run in three sets of 5 iterations each. The larger scale results are run 3 times. The time to complete IO used is the longest time used for any process from just before the open call to just after the close completes. The data sizes are solely the application data and does not include the additional annotation nor the index areas of the BP [8] file format. Removing the open and close times and including the additional data overhead would improve these results. Additional testing is in progress to better isolate the raw IO performance achieved using these techniques.

Data staging is performed using a collection of cores no larger than a small fraction of the total computational space employed. To represent this case, process counts from 64 to 512 are evaluated. To further refine the idea, two cases are tested. First, a data size of 128 MB is tested representing a data size typical for the GTC [6] plasma physics code. Second, a data size of 768 MB is tested representing a caching data staging setup. For these tests, MPI_File_write is used to write data to storage. In Figure 4, four important results are evident. First, the anecdotal report of using 3 storage targets per file is proven to work best on average. Second, the overall best performance is achieved using a single storage target per file. Third, as

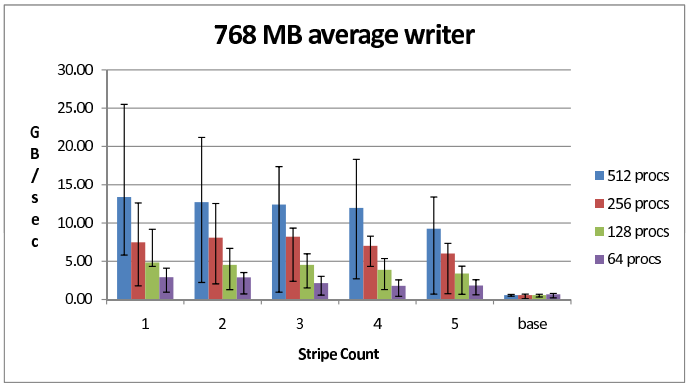

Figure 5: 768 MB per Writer

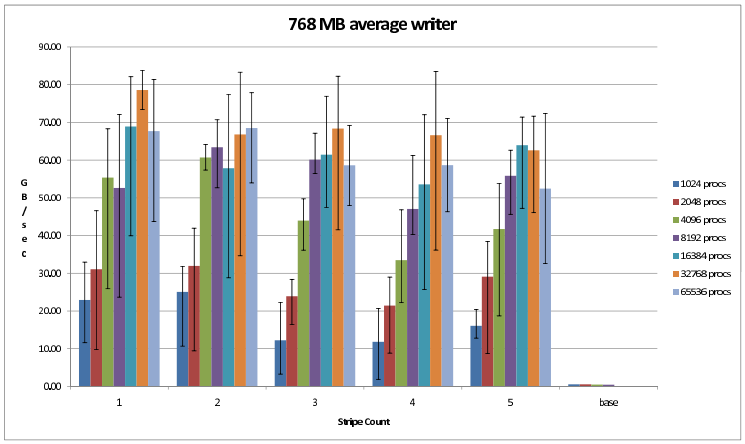

Figure 6: 768 MB per Writer

the number of storage targets per file increases, the performance steadily decreases. Fourth, the base stripe count of 4 with a single file is dramatically worse performance than any of the special tested configurations. In Figure 5, three important results are evident. First, unlike the 128 MB test above, both the average and best overall performance is with a stripe count of 1. Second, larger writes even out the average performance across different stripe counts and all at least match the 128 MB best average performance case. Third, the base case is actually slightly worse than the 128 MB case.

For both of these cases, the per-process performance is not excellent. The raw data reveals the issue. In all of these cases, the IO usually takes only 2-3 seconds. For a very small number of processes of generally 1% or fewer, the IO takes anywhere from 11 to 70 seconds to complete, skewing the results tremendously. Additional detailed measurements are required to isolate the sources of these anomalies.

4.2 Direct IO Performance

For cases where staging is not appropriate, writing directly from the compute nodes must be performed quickly. To demonstrate the efficiency of this case, various process counts from 1K (1024) to 64K (65536) are evaluated. A data size of 768 MB is used for each process. For 32K processes, this yields 24 TB of data written per output. In Figure 6, the performance picture mirrors the smaller scale 768 MB per process case. However, the top performance is 83.7 GB/sec on 32K processes, a vast improvement in data rate over the smaller scale tests. The 64K processes rates are slightly worse than the 32K results. This would suggest the system is saturated at this level with a potential reduction in performance for larger sizes. It is important to recall that the 64K processes case writes 48 TB of data at once in about 6 minutes. This is nearly the entire

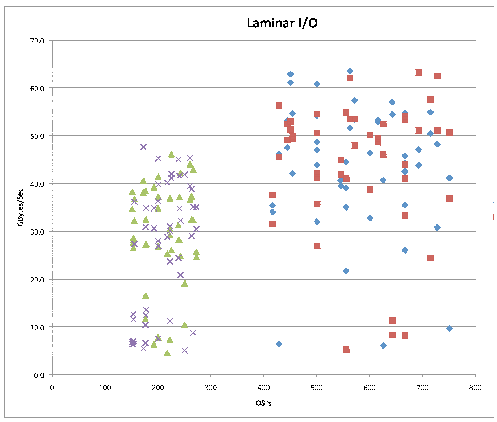

Figure 7: Aggregation Varying Stripe Count

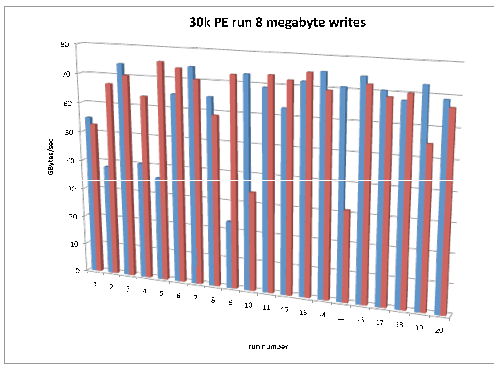

Figure 8: Aggregation 30K Cores

24 hour output of the GTC run [7], which took close to 1 hour of IO time. While the numbers are excellent and show promise, the relatively small number of runs for these cases suggests further testing is warranted to more strongly validate the results.

4.3 Aggregation

The aggregation tests consist of two sets of results. The first tests measures the prefered stripe count to use for an aggregation configuration. For the OST counts beyond 672 in figure 7, the OST selected for the writer wrapped back to the storage target 0 introducing sharing. These timings include file creation, close and fsync as part of the performance measurement. For these tests, the POSIX write call is used.

Figure 7 shows that like the scheduled IO, a stripe count of 1 performs better than the anecdotal suggested choice of 3.

Figure 8 shows the performance for a series of 30,000 core runs using both the simple and the brigade aggregation approaches. Each process has 8 megabytes to write for a total of 240 GB per run. In the routine, 442 writers are self selected to receive data as the writer. In this case there are 442 writers, which is about 65% of the OSTs on the machine. There are 29,558 processes that just pass data to a writer, or 67 passers per writer for the first 441 writers, and 11 for the 442nd writer. While the simple aggregation approach occasionally showed much greater performance than the brigade approach, the brigade approach was more consistent and frequently had better performance. This suggests that both approaches should be considered depending on the network topology employed.

5 Conclusion and Future Work

By carefully managing the use of the parallel storage system and more actively managing the data output from the application, much greater performance can be achieved. In this case, 83.7 GB/sec out of a maximum of 110 GB/sec data rates are achieved when writing from 32,768 cores in parallel. These results are demonstrated during normal machine operation with other jobs running on other parts of the machine. With these changes, the ADIOS API and this new transport method demonstrate scalability and achieve greater than 75% of peak IO performance.

For future work, the timing measurements must be made more granular to isolate the sources of variation in the timing results. Additional tests at scale must also be performed to give a better picture beyond the 2 or 3 iterations we were capable of getting so far. Varying storage target counts and data sizes is required to better understand the key factors affecting IO performance in these petascale machines.

6 References

References

[1] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, and Karsten Schwan. In *HPDC '09: Proceedings of the 18th international symposium on High performance distributed computing*, New York, NY, USA, 2009. ACM.

[2] Peter J. Braam. Lustre: a scalable high-performance file system, Nov. 2002.

[3] Philip H. Carns, Bradley W. Settlemyer, and Walter B. Ligon, III. Using server-to-server communication in parallel file systems to simplify consistency and improve performance. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–8, Piscataway, NJ, USA, 2008. IEEE Press.

[4] Karsten Schwan Hasan Abbasi, Matthew Wolf. Live data workspace: A flexible, dynamic and extensible platform for petascale applications. In *Cluster Computing*, Austin, TX, September 2007. IEEE International.

[5] HDF-5. http://hdf.ncsa.uiuc.edu/products/hdf5/index.html.

[6] S. Klasky, S. Ethier, Z. Lin, K. Martins, D. McCune, and R. Samtaney. Grid -based parallel data streaming implemented for the gyrokinetic toroidal code. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 24, Washington, DC, USA, 2003. IEEE Computer Society.

[7] Jay Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *CLADE 2008 at HPDC*, Boston, Massachusetts, June 2008. ACM.

[8] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Input/output apis and data organization for high performance scientific computing. In *In Proceedings of Petascale Data Storage Workshop 2008 at Supercomputing 2008*, 2008.

[9] Jay Lofstead, Fang Zheng, Scott Klasky, and Karsten Schwan. Adaptable, metadata rich io methods for portable high performance io. In *In Proceedings of IPDPS'09, May 25-29, Rome, Italy*, 2009.

[10] Parallel netCDF. http://trac.mcs.anl.gov/projects/parallel-netcdf.

[11] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.