Runtime I/O Re-Routing + Throttling on HPC Storage

Qing Liu, Norbert Podhorszki, Jeremy Logan, Scott Klasky
Computer Science and Mathematics Division, Oak Ridge National Laboratory

Abstract

Massively parallel storage systems are becoming more and more prevalent on HPC systems due to the emergence of a new generation of data-intensive applications. To achieve the level of I/O throughput and capacity that is demanded by data intensive applications, storage systems typically deploy a large number of storage devices (also known as LUNs or data stores). In doing so, parallel applications are allowed to access storage concurrently, and as a result, the aggregate I/O throughput can be linearly increased with the number of storage devices, reducing the application's end-to-end time. For a production system where storage devices are shared between multiple applications, contention is often a major problem leading to a significant reduction in I/O throughput. In this paper, we describe our efforts to resolve this issue in the context of HPC using a balanced re-routing + throttling approach. The proposed scheme re-routes I/O requests to a less congested storage location in a controlled manner so that write performance is improved while limiting the impact on read.

1. Introduction

The computational capabilities of high performance computing (HPC) machines are continuing to increase, which is driving the accelerating pace of scientific discovery. The Titan Cray XK7 supercomputer, the world's fastest machine as of November 2012, hosted at Oak Ridge National Laboratory has 299,008 processing cores and 18,688 GPUs, with a peak performance of approximately 20 petaflops. For HPC data-intensive applications, the volume of data generated per run is projected to grow quickly and the question of how to efficiently manage such large quantities of data becomes increasingly difficult.

Parallel storage systems have recently become a viable solution for today's scientific applications and are widely deployed on many of today's Top500 systems such as Titan, Hopper and Intrepid. These systems typically use large numbers of storage devices (e.g., 100s) to achieve the throughput and capacity demanded by large scientific applications. A recent test on Titan/Jaguar demonstrated an impressive 240 GB/sec I/O throughput over 672 storage devices [1]. Despite these strong benchmark results, we have seen significant I/O fluctuations in real production environments, observing as much as an order of magnitude variation in throughput per storage device. A root cause of such I/O variations is the interference posed by other applications running simultaneously and sharing either network or storage resources. Compared to cloud storage contention discussed in previous work [2], the interference can be at a whole different level due to the sheer amount of concurrency present in an HPC system. This can pose disastrous effect to the performance of an HPC system if left unchecked. In the worst case, for instance on Titan, one single storage device could possibly be rendered useless by an O(100,000)-core job attempting to perform I/O in a naïve manner using a single target. Using a log-structured file system (LFS) [3] can alleviate this symptom to some extent but does not completely resolve the issue for a large HPC system, as even with solid-state drive (SSD), the disk cache has difficulty accommodating data dumped from O(100,000) cores. Caching is not deemed to be the clear path forward for future HPC systems as projections indicate that the number of cores will continue to grow while memory/cache per core is expected to decrease [10]. On the other hand, I/O concurrency is critical for application performance. Although the chaining logging technique [2], which is based on LFS works extremely well in the context of cloud storage, it suffers from serialization of I/O tasks caused by controlling the sequence of logging to avoid collision between garbage collection and application output. As such, this approach inevitably reduces disk concurrency and, despite alleviating contention, it lowers overall throughput and is not a promising solution for HPC.

Meanwhile, previous work [4] tackles the contention issue by isolating applications through explicit QoS support. The devised pClock algorithm captures the bandwidth and burst requirements based upon arrival curves. This scheme attains the isolation goal very well. However, in the HPC world, it is hard to stipulate fine-grained QoS parameters for a particular application since most, if not all, applications have equal service level requirements which are "as fast as possible".

Hotspots caused by I/O contention are detrimental to parallel application performance as they lead to variations in completion times across processes. Typically parallel application processes are forced to operate in a tightly synchronized manner, which means even a process that finishes its I/O earlier is still forced to wait for its slower peers. Multiplying this variance by the number of cores, the wasted computational capacity can be far too costly to ignore, particularly for the future HPC

system where a billion-way concurrency is planned. In contrast to cloud storage, the scale of HPC storage can push the performance even closer to the edge. As an example, even averaging just one second of idle time for a billion core job would lead to O(100,000) CPU hours wasted, which is extremely costly. Furthermore, the variability in I/O time makes job time less predictable. If a job doesn't finish on schedule due to interference caused by a competing job, the entire run or the portion of the run starting from the most recent checkpoint has to be re-done. The remainder of the paper is organized as follows. Section 2 makes observations on I/O fluctuations on production systems. Section 3 discusses the design and implementation details of I/O rerouting with throttling. Section 4 presents performance evaluations along with conclusions in Section 5.

2. Observations and Motivation

In this section we present measurements collected on two high-end machines and demonstrate that I/O hotspots do exist on production storage systems. The discussion in this paper specifically targets the shared storage system, in which users share the storage resources, as contrasted with other strategies such as dedicated storage or storage with QoS guaranteed using scheduler [4][5]. We argue that the case we target is prevalent for HPC systems and the scheduling is often not a viable solution as an I/O scheduler dealing with a billion-way currency is not efficient if even possible.

The experiments were done on Titan at Oak Ridge National Laboratory and Hopper at National Energy Research Scientific Computing Center (NERSC). Each machine deploys a massively parallel storage system, which consists on the order of O(100) storage devices for a given file system scratch space. For the first experiment, we continuously write (and flush) 16 MB block to the first storage device in the system to test the perceived speed of an individual storage device. The results were collected over 100 iterations to show the severity of the throughput fluctuation over time. This fluctuation is particularly apparent on Hopper, as seen in Figure 1(b), where the peak is more than ten times higher than the lowest performance. More importantly the results indicate that the performance of the storage system is in fact bursty and instable. Therefore any offline approach that tries to use mining technique over the storage logs is likely to be quite ineffective in dealing with hotspot problems. Figure 2 shows the performance snapshot of writes to 100 storage devices on both Titan (a) and Hopper (b). It is interesting to notice that although most of the storage devices offer similar throughput, there are a few outliers that are significantly slower.

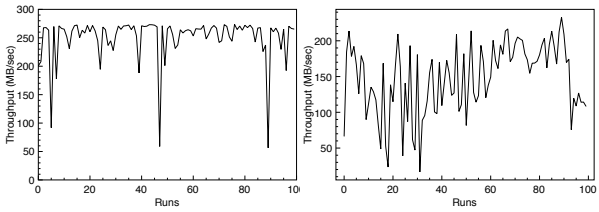

Figure 1: Performance of storage device #0 (a) Titan (b) Hopper

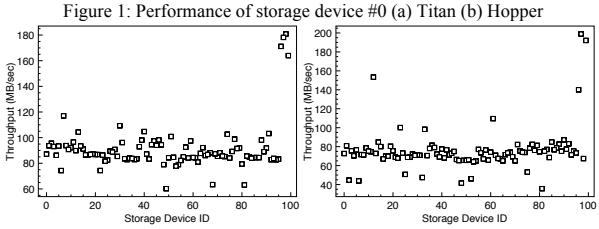

Figure 2: Snapshot of 100 storage devices #0–#99 (a) Titan (b) Hopper

2.1. Impact of I/O Imbalance to Parallel Applications

Hotspots in the storage system are much more harmful to parallel applications than serial applications. The reason is that many large-scale parallel scientific codes today use *message passing interface* (MPI) collectives to exchange the state between processes. For example in QLG2Q [6], a quantum lattice gas code, each process needs to disseminate values of ghost cells to adjacent processes before the calculation for next iteration begins. These ghost cells then provide boundary conditions for solving the objective partial differential equation (PDE). With hotspots in the storage system, each process progresses differently in time and, at some point during the run, a fast process has to sync with other processes and wait for ghost cells data. For high-end machines where there are more than hundreds of thousand cores available to applications, the speed gap between the fastest and slowest device can be disastrous. For example in the test run in Figure 2(b), the speed of storage device #98 on Hopper is five times as fast as that of storage device #48, with the difference caused by the heavy I/O activities of other users. The consequence is that CPU cycles are wasted on fast processes, which leads to overall application inefficiency.

2.2. Contributions

Our previous work on the I/O variability issue [7] tackled the problem by first looking at write optimization only, using a somewhat simplistic approach. The proposed adaptive I/O technique uses a single execution thread for both write and communication on coordinator and sub-coordinator processes. This design limits how quickly a coordinator can respond to storage load dynamics, and for small request workload, e.g., *Pixie3D small* (section 4(b)), I/O requests cannot be processed responsively, thus adaptive I/O performance in this case is on par with non-adaptive I/O. This lowers the adapt-

ability of the system in general when I/O hotspots are present. The major contributions in this paper are:

- The impact of I/O re-routing to read performance is thoroughly investigated. Due to the re-routing, each storage device receives a varying amount of data depending on how heavily it is loaded. The more congested a particular storage location is, the less likely the data will be written to that location as the result of re-routing. This has the effect of creating secondary hotspots for later reading, as some storage devices will have more of the data than others. Ultimately the effects on both write/read performance need to be addressed.
- We present the design of a new virtual messaging layer, which places the communication tasks in a separate standalone layer, thereby minimizing the impact of messaging on the application. More importantly, this design makes I/O throttling possible, which was shown to be difficult [7].
- We propose the idea of I/O throttling to achieve a compromise between write and read. This technique effectively limits the degree of I/O re-routing so that write performance can still be improved while limiting the impact on read performance.

3. I/O Re-Routing Scheme

The idea behind I/O re-routing is based upon the observed imbalance in parallel storage systems that often result in a subset of devices being more highly loaded than the others. Our proposed I/O re-routing scheme attempts to mitigate this imbalance by re-directing I/O traffic to less loaded storage locations, thereby reducing the amount of data being written to congested devices. This work is implemented as an I/O method under the ADIOS umbrella [8]. The core component of our I/O re-routing scheme is a virtual messaging layer that serves to disseminate storage state to each process that participates in I/O. Each process is attached to this messaging layer and is notified and re-directed to a new target if its current target becomes heavily congested.

3.1. Virtual Messaging Layer (VML)

The VML provides the necessary messaging capabilities to the client processes. It receives incoming I/O requests from client processes, grants permission to do I/O, or re-directs a request to a more desirable storage location. It internally uses short messages to disseminate storage state to each participating group so that the occurrence of congestion can be notified quickly and acted upon. The messages are exchanged in a quite reactive fashion, i.e., only when a storage device becomes idle, its controller initiates appropriate re-routing messages. The incurred messaging overhead is quite minimal as the high-speed interconnects today, for instance, the Cray Gemini network on Titan poses extremely low latency (~2.5μs), which is negligible compared to disk I/O fluctuations.

Figure 3 illustrates the overall architecture of I/O re-routing framework. Here a group represents a set of processes that share a common initial storage target location to write. This target may change when the ensuing I/O re-routing happens. In this case, the re-routed process will leave its original group and join a new group where I/O load is expected to be lighter. VML uses a group-based two-level control framework to facilitate message passing. To enable the message passing between groups, a *global coordinator* (GC) is selected for all groups and a *sub-coordinator* (SC) is selected to arbitrate the messaging between the SC and the individual processes (P). The reason for the two-level design is to make communication more scalable so that the GC is not required to handle messages from every process, which would greatly limit scalability. Implementation wise, SC/GC can be either an in-kernel device driver or a thread launched alongside applications. In this work, the latter scheme is used and VML is attached to each process as a separate pthread, which allows an application and VML to run in parallel so that the execution of one doesn't block the other.

Before diving into details, we briefly introduce the notations used in this paper. A process is denoted as P_i where $0 \le i \le N-1$ and N is the total number of processes. The number of storage devices is M and, without loss of generality, we also assume N is multiple of M. Therefore the number of processes that each group has is N/M and hence, for group G_i, the initial process set it contains is $[P_{N/M * i}, P_{N/M * (i+1)-1}]$. The SC_i is the sub-coordinator for group G_i, which writes exclusively to the i-th storage device SD_i. Note SC_i can be attached to one of the processes in the group, e.g., $P_{N/M * i}$.

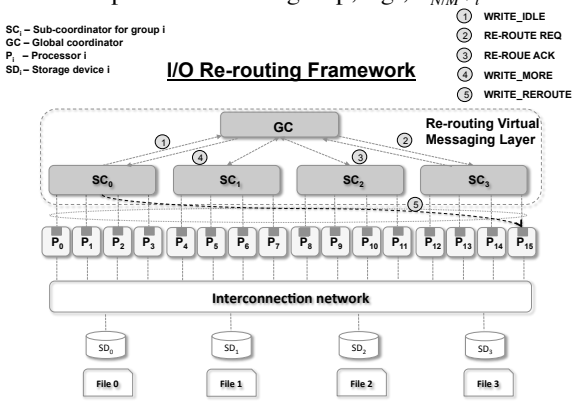

Figure 3: I/O Re-Routing framework

3.2. I/O Re-Routing

The I/O re-routing phase is jumpstarted by SC_i issuing WRITE_IDLE message to GC, indicating group G_i (and hence SD_i) has finished all its I/O tasks and is in idle state. This only occurs when all pending processes within G_i finish writing. GC is a central controller

which keeps track of the state of all storage locations and upon receiving a WRITE_IDLE message, *GC* updates the state of G_i to *idle* and searches for a group that is in *busy* state. If group G_j is found, *GC* then initiates re-routing process via sending RE-ROUTE REQ to SC_j to request offloading portion of its I/O load. When this request is acknowledged by RE-ROUTE ACK, GC constructs a WRITE_MORE message with its payload carrying re-routed process rank, P_k, and re-directs process P_k to write to the new storage device (via WRITE RE-ROUTE). An example of the messaging flow is illustrated in Figure 3. The re-routing phase is ended when no group is in busy state, and when all I/O requests are finished (i.e., when file is closed) VML will be de-attached from each process and then released.

3.3. I/O Re-Routing + Throttling

The I/O re-routing technique can maximize write performance on a busy system and effectively avoid storage hotspots. As a result, the degree of I/O variability perceived by applications can be mitigated. However, this scheme is aggressive in the sense that it places a larger burden on a fast storage device, and the result is that a varying amount of data is written to each storage device, depending on the degree of imbalance experienced. This is usually acceptable for checkpoint output because this type of I/O operation is write-dominated and most checkpoint files are unlikely to be read back in the future. However, for the diagnostic data which will be read multiple times, the overly aggressive nature of re-routing write operations can cause a secondary load imbalance for reading in terms of data size, i.e., some storage devices have significantly more data to read than the others. To address this, we apply a ***throttling*** technique that limits how much data may be re-routed during writing, thus avoiding hotspots while lessening the effect of re-routing on read performance. To achieve this, we introduce the concept of *throttling factor* (TF) for each SC, which is defined as the ratio of the amount of data re-routed to the associated group versus the amount of data that originally belonged to the group *and* has been written (i.e., local data). For each storage device, TF essentially limits the size of new I/O requests that can be accepted inbound by an SC. If the current ratio is no greater than TF, the re-routed request will be granted. Otherwise, the storage device has accommodated too many re-routed requests and the new request will be rejected. In that case GC will try the next available idle storage location. Overall, this scheme limits how much data a fast location can take and balance the data across storage locations. Note that a slow storage device will offload a portion of its I/O requests, and the resulting data written to that location is therefore reduced. As such it will allow fewer requests to be accepted inbound, as compared to a fast storage location. We argue that this is a reasonable strategy as a slow storage is likely to continue to be slow for a short period of time (at least for the duration of the run) and, therefore, less re-routing should be allowed. This assumption fits particularly well to HPC applications as they follow the pattern of:

Computation→Synchronization→I/O→Synchronization →Computation→ Synchronization→I/O ……

This suggests that a storage location undergoing heavy I/O traffic is likely to be hit by a similar burst of traffic in the next cycle as well (also shown in Figure 1).

4. Performance Evaluation

To gauge the effectiveness of the I/O re-routing framework, we ran a synthetic benchmark and Pixie3D I/O kernel on both Titan and Hopper. Titan is a Cray XK7 machine and has 18,688 compute nodes in addition to dedicated login/service nodes. Each compute node contains a sixteen core 2.2GHz AMD Opteron processors with 2GB DDR3 memory/core, and a Gemini interconnect. The work was performed during the final phase of Titan acceptance test and 9718 compute nodes were available when the experiments were conducted. We ran the benchmark and Pixie3D on Titan Lustre file system *widow1*, which has 336 storage devices [1]. Hopper is a Cray XE6 machine and has 6,384 compute nodes each consisting of two 12-core AMD MagnyCours 2.1 GHz processors, with 2GB DDR3 memory per core. We conducted the Hopper runs on the Lustre SCRATCH space, with a total of 156 storage devices.

The majority of the runs are done with artificial noise injected into the system in order to mimic the random effect of I/O interference from other users. This noise is generated by a parallel program that continually writes a 16MB block to selected storage locations. Here we experimented with two noise setups: *light interference* and *heavy interference*, to see how the I/O re-routing scheme adapts to the external noise. In light interference, 16 processes write to storage device 0 whereas in heavy interference 64 processes write to storage 0-3 with 4 processes/storage. Light interference was injected for each test run unless otherwise noted.

A. Synthetic Benchmark

Our synthetic benchmark is a simple parallel code that writes and reads a configurable size chunk of data. Figure 4 shows the total I/O time for writing a 2 MB chunk from 64 to 4096 processes on Titan and Hopper. Each data point in the plot is the average result of 20 consecutive runs. Clearly, I/O re-routing results in a much improved write performance compared to static I/O (i.e., re-routing is not used). The gain is particularly noticeable at higher core counts, providing a 67% reduction of write time on Titan and 33% on Hopper at 4,096 cores. Figure 5 further evaluates the sensitivity of the performance to TF value under light and heavy interference using 1024-cores. An interesting insight is

that a relatively small TF is sufficient to achieve a large margin. Enlarging TF beyond a certain point doesn't seem to provide further performance improvement. However, as interference gets heavier, enlarging TF is necessary to further lower the I/O time. One caveat is that a larger TF will result in a higher degree of data imbalance, and is therefore less efficient for reading. Figure 6 shows the read time of 20 runs from the data generated previously under heavy interference in Figure 5. Clearly, although static I/O is bad for write performance, it does yield the best performance for reading, as data is evenly distributed across storage devices. For example, in Figure 6, static I/O is 52% better than re-routing with TF set to 0.1. As TF increases, the read time also increases. In reality, to achieve a balance between write and read, one needs to set a relatively small TF, whose optimal value depends on the degree of interferences in the system and is left for future study.

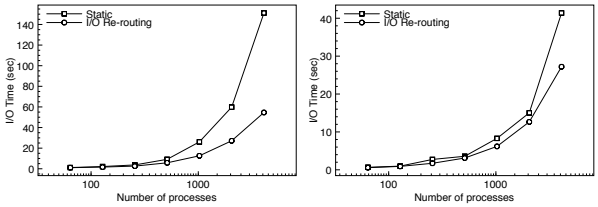

Figure 4: Write performance of static vs. I/O re-routing (a) Titan (b) Hopper

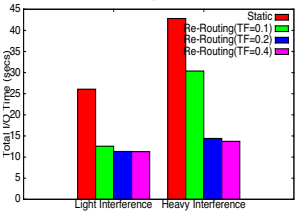

Figure 5: Sensitivity of TF

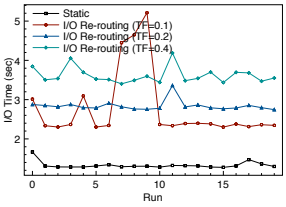

Figure 6: Read performance

Figure 7 shows the write performance observed during a 2-hour window on Titan and Hopper, without any artificial noise being injected. This gives an indication of how I/O re-routing works in a production settings where interferences are from random users. The static I/O and I/O re-routing runs were interleaved to achieve maximum fairness. Here I/O re-routing shows stabler as well as lower I/O times than static I/O. Note that HDF5 file format is used here due to its wide adoption to science community, and the self-describing format makes writing much more costly than reading.

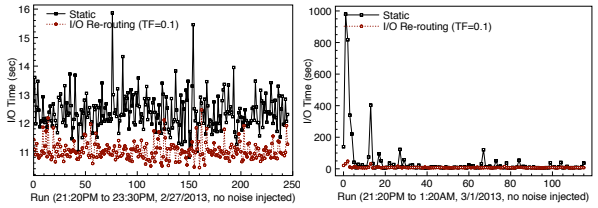

Figure 7: Write performance without interference injected (a) Titan (b) Hopper

B. Pixie3D

Pixie3D [9] is an extended Magneto-Hydro-Dynamic (MHD) code that solves extended MHD equations using fully implicit Newton-Krylov algorithms. The output contains eight 3D cubes using 3D domain decomposition. The size of each 3D array is typically sized 256x256x256 (hero), 128x128x128 (large), 64x64x64 (medium) and 32x32x32 (small). Due to space limits, this paper only presents the 32x32x32 setup of Pixie3D at 1024 core run. Similar to previous results, static I/O exhibits a much longer I/O time, particularly under heavy interference, 290 secs vs. 125 secs of re-routing, a 57% improvement, with TF set to 0.2, see Figure 8.

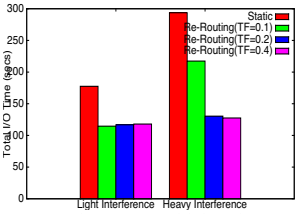

Figure 8: Sensitivity of TF

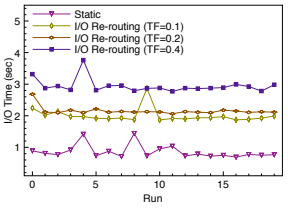

Figure 9: Read performance

5. Conclusion

This paper attempts to resolve the I/O contention issue in the context of HPC storage, where a large number of cores (i.e., interfering sources) are present. We propose a balanced re-routing + throttling approach to alleviate the contention. The performance results indicate that the scheme works well for both our synthetic benchmark and the Pixie3D code.

Acknowledgement

The authors would like to thank our shepherd Nohhyun Park from Cloud Physics as well as anonymous reviewers for the valuable suggestions and the Department of Energy Office of Science for the sponsorship.

REFERENCES

[1] Galen M. Shipman, *et al.*, Lessons Learned in Deploying the World's Largest Scale Lustre File System, The 52nd Cray User Group Conference (CUG '10), May 2010.

[2] Ji Yong Shin, *et al.*, Gecko: A Contention-Oblivious Design for Cloud Storage, USENIX HotStorage'12, Boston, MA, June 2012.

[3] SENBLUM, M., *et al.*, The design and implementation of a log-structured file system. ACM Trans. on Comp. Sys. 10 (Feb 1992).

[4] A. Gulati, A. Merchant, P. Varman, P-Clock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems, ACM SIGMETRICS'07, June, 2007.

[5] Xuechen Zhang, *et al.*, "QoS Support for End Users of I/O-intensive Applications using Shared Storage Systems", ACM/IEEE SC'13, Seattle, WA, November 2011.

[6] G. Vahala, *et al.*, "Unitary Qubit Lattice Simulations of Multiscale Phenomena in Quantum Turbulence", ACM/IEEE SC'11, Seattle, WA, November 2011.

[7] Jay Lofstead, *et al.*, "Managing Variability in the IO Performance of Petascale Storage Systems". ACM/IEEE SC'10, New Orleans, LA. November 2010.

[8] ADIOS, https://www.olcf.ornl.gov/center-projects/adios/

[9] L. Chacon. A non-staggered, conservative, VxB = 0, finite-volume scheme for 3D implicit extended magnetohydrodynamics in curvilinear geometries. Computer Physics Communications, 163:143–171, Nov. 2004

[10] D. Chavarria-Miranda, *et al.*, "Global Futures: A Multithreaded Execution Model for Global Arrays-based Applications", IEEE/ACM CCGrid'12, Ottawa, Canada, May, 2012.