Layout-Aware I/O Scheduling for Terabits Data Movement

Youngjae Kim, Scott Atchley, Geoffroy R. Vallee, and Galen M. Shipman
Oak Ridge National Laboratory
Oak Ridge, TN 37831, USA
{*kimy1, atchleyes, valleegr, gshipman*}*@ornl.gov*

***Abstract*—Many science facilities, such as the Department of Energy's Leadership Computing Facilities and experimental facilities including the Spallation Neutron Source, Stanford Linear Accelerator Center, and Advanced Photon Source, produce massive amounts of experimental and simulation data. These data are often shared among the facilities and with collaborating institutions. Moving large datasets over the wide-area network (WAN) is a major problem inhibiting collaboration. Next-generation, terabit-networks will help alleviate the problem, however, the parallel storage systems on the end-system hosts at these institutions can become a bottleneck for terabit data movement. The parallel storage system (PFS) is shared by simulation systems, experimental systems, analysis and visualization clusters, in addition to wide-area data movers. These competing uses often induce temporary, but significant, I/O load imbalances on the storage system, which impact the performance of all the users. The problem is a serious concern because some resources are more expensive (e.g. super computers) or have time-critical deadlines (e.g. experimental data from a light source), but parallel file systems handle all requests fairly even if some storage servers are under heavy load. This paper investigates the problem of competing workloads accessing the parallel file system and how the performance of wide-area data movement can be improved in these environments. First, we study the I/O load imbalance problems using actual I/O performance data collected from the Spider storage system at the Oak Ridge Leadership Computing Facility. Second, we present I/O optimization solutions with layout-awareness on end-system hosts for bulk data movement. With our evaluation, we show that our I/O optimization techniques can avoid the I/O congested disk groups, improving storage I/O times on parallel storage systems for terabit data movement.**

***Keywords*-Storage Systems, I/O Scheduling, Networking**

I. Introduction

Many science facilities produce a vast amount of experimental and simulation data. Several leadership computing facilities such as the Oak Ridge Leadership Computing Facility (OLCF), the Argonne Leadership Computing Facility (ALCF), and the National Energy Research Scientific Computing (NERSC) generate hundreds of petabytes per year of simulation data and are projected to generate in excess of 1 exabyte per year by 2018. Moreover, other scientific user facilities and data centers such as the SNS, a major neutron facility, or the Atmospheric Radiation Measurement (ARM) program generate tremendous amounts of experimental and/or observational data.

These data sets do not exist in isolation. Scientists and their collaborators may have access to additional resources at multiple facilities and/or universities. Scientists may use these additional resources for further analysis and visualization or they may use experimental results to validate ongoing simulations. Moving the data between geographically dispersed organizations is necessary to further their research. Some examples of large collaborations include: one OLCF petascale simulation needs nuclear interaction datasets processed at NERSC; the ALCF runs a climate simulation and validates the simulation results with climate observation data sets at ORNL data centers.

In addition to the growing size of data sets, network operators are increasing the capabilities of the network. DOE's Energy Sciences Network (ESnet), for example, has upgraded its network to 100 Gb/s between many DOE facilities. Future deployments will most likely support 400 Gb/s followed by 1 Tb/s throughput. Even with these networks, data sharing will remain difficult and will require advanced end-to-end optimization techniques to coordinate network and storage technologies to utilize the technology advances to the fullest extent. Even as network and storage technologies continue to advance, the storage-to-network I/O constraint will likely continue to be a major challenge in our terabit data transfer architecture.

Parallel file systems have been a widely adapted solution for scientific applications to support both high performance I/O and large data sets. Typically, these large scale storage systems use tens to hundreds of storage servers, each with tens to hundreds of disks, to improve scalability of performance and capacity. These file systems stores individual files over subsets of disks to improve single file performance. The Oak Ridge Leadership Computing Facility's *Spider* file system serves as a center-wide storage resource for all compute systems including one of the world's fastest supercomputers, Titan. The Spider II system is designed to provide 32PB of capacity with tens of thousands of spindles at an aggregate transfer rate of 1 TB/s. Spider II can only achieve that rate, however, when all the spindles operate at their maximum speed for ideal, sequential I/O patterns. As this is a shared resource where multiple jobs can share the same storage groups, typically throughputs are less. Hotspots (i.e. congestion) occur within some of the disk volume groups, which causes an I/O load imbalance over the disk

978-1-4799-1293-3/13/$31.00 ©2013 IEEE

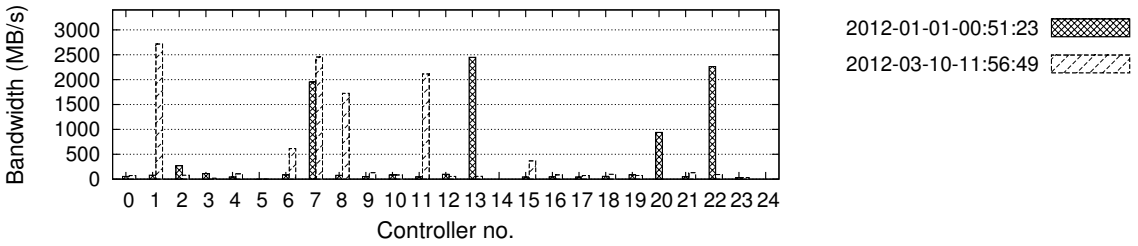


Figure 1. Snapshot of individual controller's bandwidth for two different times. We show the statistics of performance data for each day above with (Min, Max, Avg, STD, Date). (0.0, 2449.0, 373.2, 724.6, 2012-01-01), (0.0, 2713.0, 465.5, 822.6, 2012-03-10).

volume groups.

I/O load imbalance is a serious problem in parallel storage systems [6], [5]. We empirically observed the bandwidth imbalance of the 24 RAID controllers for a quarter partition of the Spider storage system at Oak Ridge National Laboratory. Figure 1 presents the snapshots of the controllers' bandwidths at two different times. We clearly see that a few of controllers are overloaded while most are not. The details of how we collected the data will be explained later in Section III. One slow disk volume can significantly decrease the performance of bulk synchronous I/O workloads. Most parallel file systems balance I/O loads in terms of space, to allow the disk volumes usage to grow at the same rate, with the constraint that individual files might only use a subset of storage servers. As a result, some of disk volumes are overloaded by a large number of data read and write requests. Caching and buffering can alleviate the I/O load imbalance problem to some degree, but they cannot solve the issue completely. Prediction based I/O rescheduling can also alleviate the issues of overloading disk volumes to some extent, however, in HPC systems, it is very difficult to predict I/O requests and their disk layouts.

With these motivations, this work makes the following contributions:

- This work investigates the problems of optimizing end-to-end data transfers in a dynamic operating environment, where I/O loads to disk volumes periodically change, often dramatically. To identify the I/O load imbalance problems in real world systems, we analyze the I/O performance data collected from each disk controller in the OLCF Spider storage system.
- To avoid the congested disk groups for terabit data movement, we propose I/O scheduling algorithms with layout-awareness on source and sink hosts. Our layout-aware scheduling algorithms can mitigate the storage-to-network I/O imbalance, a major problem in our terabit data transfer architecture. These algorithms adapt dynamically to changing I/O loads on the storage systems by avoiding overloaded disk volumes while striving to maximize available, uncongested I/O bandwidth.

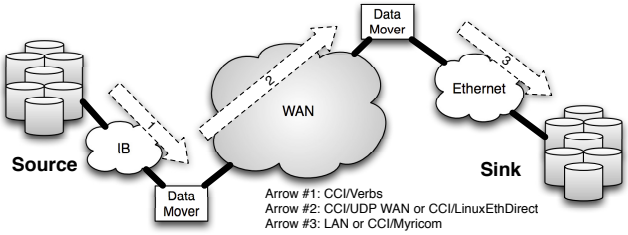


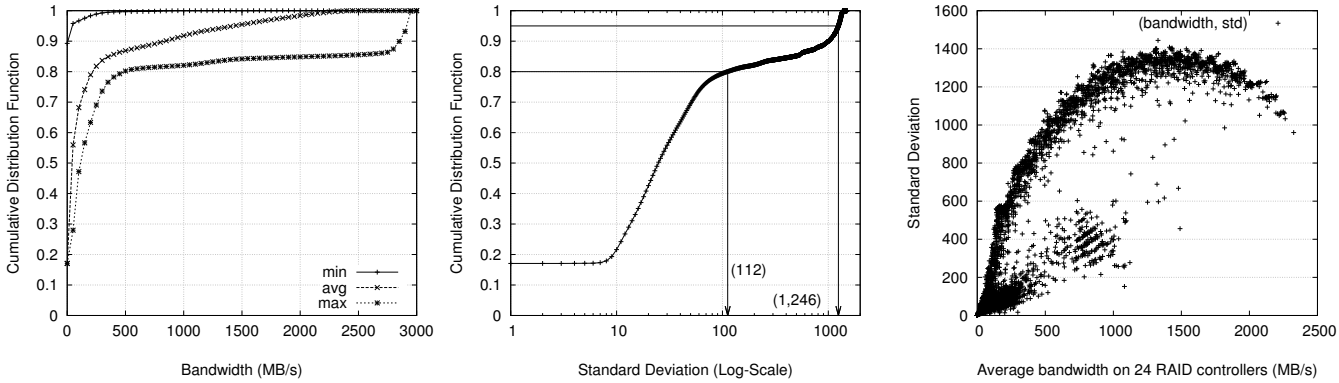
Figure 2. Data Movement Flow Diagram.

The rest of this paper is organized as follows. Section II presents an overview of terabit data movement and discusses related works for bulk data movement. Section III investigates the problems of load imbalance over disk volume groups in the shared storage sytsems. Section IV presents our design and implementation on I/O scheduling techniques for PFS. We show our evaluation results in Section V and conclude in Section VI.

II. Overview for Bulk Data Movement

When moving between two PFS at separate sites, the data will traverse one or more networks. Over the wide-area network (WAN), the data will travel via high-bandwidth networks such as DOE's ESnet and Internet2. If a PFS is not directly connected to the WAN, the data will additionally transit one or more local networks. Some of these networks may provide zero-copy, OS-bypass interfaces and others only support the standard Sockets interface. In Figure 2, we illustrate data moving from a PFS over an InfiniBand (IB) network to a data mover connected to the WAN. The data arrives at the remote site at another data mover which forwards the data over the local Ethernet network. The Common Communication Interface (CCI)[2] is used to enable zero-copy, OS-bypass when supported by the network to improve performance. CCI provides message (MSG) and remote memory access (RMA) semantics. We use MSGs for control and RMA for bulk data movement. Also, the data movers act as routers between different networks.

Bulk Data Movement: There have been many prior studies on the design and implementation of bulk data movement



(a) Bandwidth (b) Standard Deviation (c) Bandwidth and Dispersion Relationship

Figure 3. Analysis results for I/O load imbalance on the Spider storage system. We analyzed bandwidth data collected from 24 RAID controllers of the Spider storage system for three months from Jan. to Mar. in 2012.

framework [1], [3], [11], [10] and their optimization in wide-area networks. GridFTP [1], provided by Globus toolkit, extends the standard File Transfer Protocol (FTP), and provides high speed, reliable, and secure data transfer. BBCP [3] is anther data transfer utility for moving large files securely, and quickly using multiple streams. These tools are useful for moving large data faster and securely from source host to remote host over the network, however little work has been done to optimize I/O performance for PFS. XDD [11] optimizes the disk I/O performance; enabling file access with direct I/Os and multiple thread for parallelism, and varying file offset ordering improve I/O access times. However, there are several key differences when compared to our work. First, our work, in particular, focuses on developing congestion-aware I/O optimization techniques on data source and sink hosts in a shared storage environment rather than optimization for dedicated storage systems. Second, we address the challenges of integrating our I/O optimization algorithms to resolve the I/O impedance mismatch between storage and network using the Common Communication Interface (CCI) [2]. Third, our work focuses on sourcing and sinking to parallel file systems where data is distributed among many storage servers.

III. I/O Load Imbalance on the Storage System

In this section, we present our analysis of actual measurements on a real production system at Oak Ridge National Laboratory. Our analysis clearly illustrates the I/O load imbalance (LI) problems in the storage systems for HPC workloads. We are interested in storage systems that that run parallel file systems for multiple scientific applications running concurrently on different sets of compute nodes. We observe the presence of intermittently congested storage servers and the duration of this congestion. Lastly, we motivate the need for an implementation of layout-aware I/O optimization at end systems for bulk data movement.

A. Data Collection at Spider

Spider is a Lustre-based storage cluster of 96 DDN S2A9900 RAID controllers (henceforth referred to as a controller) providing an aggregate capacity of over 10 petabytes (PB) from 13,440 1-terabyte SATA drives [12], [4]. For analysis, we characterize I/O statistics for the data we collected from the controllers. The controllers have a custom API for querying performance and status information over the network. A custom daemon utility [8] periodically polls the controllers for data and stores the collected results in a MySQL database. Among the various I/O performance metrics, our primary interest is studying the I/O load imbalance. To do so, we measure the bandwidth and latency for reads and writes every two seconds. We studied this workload for 24 of the controllers over a period of three months (January–March 2012). This partition includes a quarter of our total storage system (in terms of capacity and performance) and its workload is representative of our overall workload.

B. Bandwdith Analysis

The I/O bandwidth distribution helps us understand the I/O utilization and requirements of our scientific workloads. Figure 3(a) shows the controller utilization in terms of bandwidth. We generated the CDF (Cumulative Distribution Function) plot of the minimum, average, and maximum bandwidth data. Each bandwidth is the sum of the bandwidth for reads and writes as well as the internal bandwidth taken by background services, such as disk scrubbing for latent sector errors [7]. From the CDF plot, we extract the 95th percentile of the minimum, average, and maximum bandwidth; 21MB/s, 1,430MB/s, and 2,908MB/s respectively. This observation implies that the bandwidth distribution is highly likely to follow a heavy, long-tail distribution. The heavy-tail distribution can be found in many natural phenomena such as stock market crashes and earthquakes, which do not occur often but are clustered when they

occur. Similarly, our observation implies that an individual controller would receive a burst of I/O demands in a very short time, overloading the controller. However, the CDF plot in 3(a) is not enough to explain I/O load imbalance on the storage system.

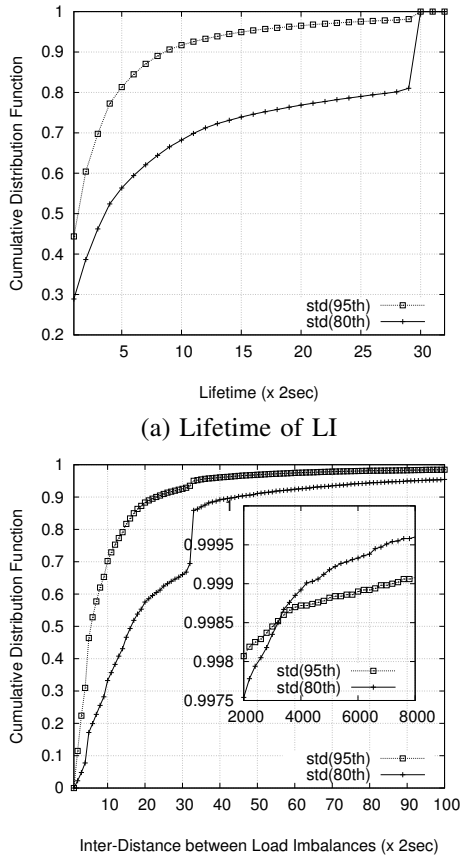
C. Load Imbalance Analysis with Bandwidth

One of the metrics for evaluating I/O load imbalance is the standard deviation of the controllers' bandwidths. For a given time period i, the instant standard deviation, std_i is calculated by $std_i = \sqrt{\frac{\sum(x_{i,j}-\bar{x})^2}{n-1}}$ where j is an index for controller. If all the controllers' I/O loads are evenly distribution at time i, the standard deviation would be very low, otherwise, it should be high. Therefore, the higher the standard deviation is, the more the I/O load imbalance is. Figure 3(b) shows the CDF plot of the standard deviation of bandwidth of the controllers. Similar to the observation for bandwidth distribution, the standard deviation plot shows a long tail distribution. For example, we observe that the 80th percentile on the CDF plot is 112 MB/s and the 95th percentile on the CDF plot 1,246 MB/s. This indicates that, over the three months, that 80% of the time the standard deviation was less than than 112 MB/s, which is reasonably low.

Figure 3(c) presents a dispersion plot of bandwidth with standard deviation. The region with `std<200` accounts for about 82% of the measurements during the three months (refer to Figure 3(b)), which is fairly balanced compared to the rest. However, the rest on the region with `std>200` shows significantly high deviation of bandwidth versus the region below 200. We observe that the standard deviation increases with respect to bandwidth. It shows either a log linear increase or a linear increase, which implies that higher bandwidth leads to more load imbalance. Interestingly, the standard deviation decreases after the bandwidth is around 1,000MB/s. It implies that many controllers start to become busy all together, reducing the I/O load imbalance.

D. Lifetime and Inter-Distance of Load Imblance

The lifetime distribution of load imbalance period is essential to understanding and optimizing storage and file system performance. Above, we use the controllers' standard deviation as the imbalance metric. For a given time period i, the instant standard deviation is smaller than the threshold, then the storage systems can be said to be fairly balanced, otherwise, the system is poorly balanced. We use two thresholds, which are 112 and 1,246 standard deviations corresponding to 80th and 95th percentiles in Figure 3(b). Figure 4(a) presents the lifetime distribution of load imbalance. Interestingly, we observe that the higher threshold is the shorter lifetime. For example, for the 90th percentile, the lifetime for std 95th is around 16 sec, whereas that for std(80th) is around 58 sec. The load imbalance does not continue over 60 seconds in our data analysis.



(a) Lifetime of LI

(b) Distance between LIs

Figure 4. Lifetime and distance distribution of load imbalance.

Inter-distance analysis provides an estimate of time between load imbalance periods. Interestingly, we see that the short lifetime for std 95th shows a smaller inter-distance than for std 80th for 99% of the time, which means that once the system is highly imbalanced, the system is likely to remain imbalanced. However, as we see from the zoom-in plot, two lines are crossed around at 3000–4000 seconds. This indicates that the 95th data is more bursty than the 80th data in terms of I/O load imbalance duration. Note that the 95th shows a longer tail distribution than the 80th.

E. Impact of I/O Load Imbalance Problem

When data are moved from source to sink over the network the data that resides in the storage system must be read through the parallel file systems, buffered in the application's memory region, and then shipped over the network. In a PFS, a file is striped over multiple object storage nodes. The highest throughput can only be achieved if none of the storage nodes are hotspots. However, unfortunately we have seen that hot spots often happen, and those hot spots create I/O load imbalance problems on storage systems. When a file is transferred to the sink node, the file needs to be

striped and stored in multiple object storage nodes as well. Similar to the impact of hot spots on file reads at source, hot spots can delay write completions of objects, increasing a file-write completion time. When hot spots occur, I/O requests be serviced responsively. Therefore, in our design I/O schedulers at source and sink use algorithms that avoid these hot spots.

IV. I/O Scheduling for Terabit Data Movement

We observed that storage systems have had intermittent congestion on a subset of storage servers in Section III. In this section, we present our design and implementation for efficient I/O scheduling at the data source and sink hosts for bulk data movement.

A. Design Goal

The source algorithm optimizes I/O accesses at the data source and the sink algorithm optimizes them at the data sink. Source and sink algorithms are designed to exploit high I/O parallelism available in the PFS, and adapt to congestion on the storage servers to minimize the impact on I/O operations. Our design goal is that *less congested servers progress faster for I/O accesses*. We implement several policies for comparison to evaluate the effectiveness of our algorithms with layout-awareness at data source and sink hosts in our bulk data movement architecture. Our scheduling algorithms can automatically detect congested servers and avoid accessing them, while sustaining I/O throughputs similar to the uncongested case.

B. Layout-Aware Source Algorithm

We implement two policies, *naive* and *layout-aware* algorithms. The naive algorithm exploits I/O parallelism in the PFS with multiple worker threads, and uses a file as its data access granularity. A global queue maintains a list of files that will be read. In this policy, a complete file is assigned to each thread, and each thread works on the file until the file is read. After completing the file, the next file task is dispatched from the queue. It ignores file layout information, thus there is no overhead involving metadata access on metadata servers. Note that in PFS, large files are broken up into to chunks and stored on multiple storage servers. The metadata is stored on one or more metadata servers depending on the PFS and the metadata access requires a network round-trip.

The layout-aware algorithm uses a *chunk* as a its data access unit. A file is segmented to chunks according to stripe size. A chunk is assigned to each thread, unlike the naive implementation where a file is assigned to each thread. A thread works on a chunk of a file, and it may work on a chunk of a different file. It needs the stripe information, such as stripe size, and width, thus requiring an additional metadata access to determine stripe information. In our evaluation, we found that this metadata overhead is negligible, compared to the performance gain by this algorithm. It implements a *server congestion detection and avoidance algorithm*, to avoid intermittently congested storage servers. As such, each storage server has a separate queue and each file stripe information is input to the appropriate server queue. Threads select the next server queue in round-robin and retrieve the stripe information. If another thread is already accessing the server, it skips and procedes to the next server. Eventually, it could achieve performance gains by allowing more progress on fast (uncongested) servers than on slow (congested) servers.

We implement our algorithms using pthreads in C and test using the Lustre PFS. The naive algorithm implements a global queue and data access at the granularity of a file, whereas data layout-aware algorithm maintains a separate queue for each of the servers. In the layout-aware algorithm, the data access unit is a chunk (i.e., an object in Lustre). The layout-aware implementation uses a Lustre utility library, `get_file_info()` to find object information of a file such as stripe size, width, and object layout. It uses the POSIX `pread()` to access specific chunks on a file. It implements a server-congestion-detection algorithm, which is a threshold-based throttling mechanism. A thread reads a chunk on a specific region on a file from its appropriate server and records the chunk read time, and computes an average of multiple chunk read times during a pre-set time window time (`W`). If the chunk read time on average during `W` is greater than the pre-set threshold value (`T`), then the server is marked congested. When marked congested, the algorithm tells the next `M` number of threads to skip this server's queue.

C. Layout-Aware Sink Algorithm

As with the source, we implement two sink policies, *file-per-server* and *layout-aware* algorithms. The file-per-server algorithm exploits I/O parallelism for parallel file systems using a multi-thread design. It assigns the threads to servers in round-robin, and the thread begins to operate with the server at the granularity of a file. A server that a thread will operate on is fixed, so it cannot avoid the servers with congestion. In contrast, the layout-aware algorithm implements a congestion detection that detects and avoids congested servers and maintains a list of uncongested servers for writes. It also uses a multi-thread design for I/O parallelism, however, different from the file per server policy, it maintains a thread pool, and the association between threads and servers is not fixed. Uncongested storage servers can store more data than congested servers. In the current implementation, when a file is written on the PFS, the file is not striped. We intentionally set the stripe width of a file to one such that it can minimize I/O interference between threads and because the current Lustre implementation does not allow the user to specify the storage servers. Both of our policies use a file as their data access granularity, however,

Workloads	Test Workloads			Overloading Workloads		
	File Size (MB)	Files (#)	Client Threads (#)	Overloading Threads (#)	File Size (MB)	Access Pattern
W(src)	20	1,000	4	2	256	RR
W(sink)	20	5,000	8	2	256	Fixed

Table I
WORKLOADS AND TEST CASES. RR MEANS ROUND-ROBIN, AND FIXED MEANS THAT ONLY ONE OST IS OVERLOADED BY THE OVERLOADING WORKLOADS.

a chunk-level implementation will provide a more flexible design in locating the servers for writes, which is a part of our future work.

V. EVALUATION

In this section, we present our experimental results to demonstrate the effectiveness of data layout-aware algorithms.

A. Experimental Setup

We evaluate our algorithms using a NetApp e5424 storage system, which has 24 SAS 10K RPM 500MB drives. We created eight logical volumes drives such that each volume is configured in RAID-0 with three of the drives. We installed Lustre 1.8 with two OSS servers with 4 OSTs each and one MDS server. Each of the OSTs manages one of the RAID volumes. All hosts run Linux kernel 2.6.18-308.4.1.el5 and are connected via IB QDR (40Gb/sec).

B. Workloads

In evaluating algorithms, we use different workloads and test cases for source and sink algorithms to cover a wide spectrum of workloads. The descriptions of these are described in Table I. In the table, *RR* means round-robin, and *Fixed* means that only one OST is overloaded by the competing workload. To generate the imbalanced loads over the OSTs, we assigned two hosts to run two overloading I/O threads. For W(src), each thread creates 256MB files on a Lustre-mounted directory, such that a file is created on the first OST, then the next file is created on the next OST, and so on. Thus, files are created in a round-robin fashion by these two threads. For W(sink), both threads create 256MB files on a particular OST (OST5 in our evaluation). For sink algorithm evaluation, prior to testing, we created files in a Lustre-mounted directory with a default setting of stripe count and size (4 and 1MB respectively), such that all disk volumes could serve the same amount of data. We minimize the caching effect on our evaluation by clearing the caches prior to each measurement.

C. Results

To fairly evaluate each algorithm's performance, we ensured that storage server bandwidth is not over-provisioned compared to network bandwidth (i.e., the network would not be the bottleneck). For storage bandwidth, we ran block-level I/O benchmarks [9] on two hosts to eight disk volumes in parallel with 1M sequential I/O streams on each benchmark with the highest queue depth. Figure 5 shows the results on comparing network and storage I/O bandwidths. The IB bandwidth increases as the message size increases, and it reaches about 3.2GB/s, whereas the I/O bandwidth is measured around 1.9GB/s at the most.

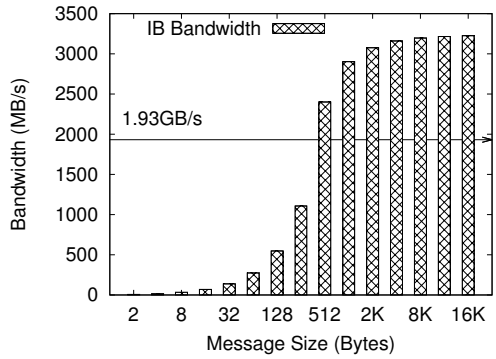
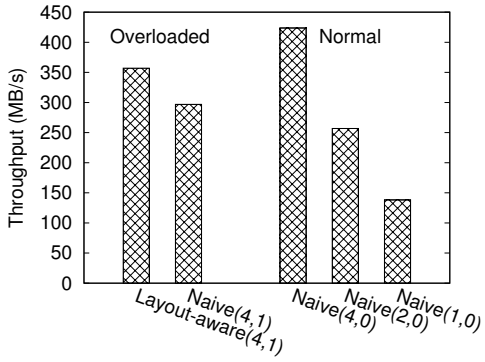
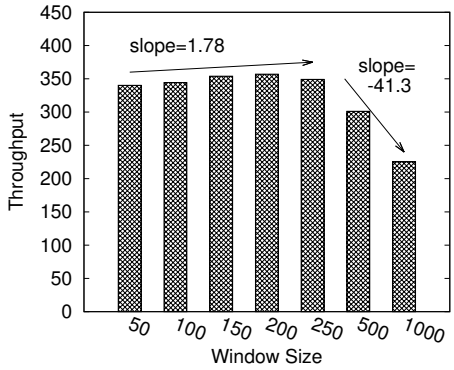


Figure 5. IB bandwidth vs. storage bandwidth. Max sequential I/O bandwidth on the storage is shown at 1.93 GB/s.

Figure 6(a) shows the performance improvement by the source algorithm with layout-awareness for W(src) in Table I. When none of the disk volumes are overloaded by I/Os, we observe that the aggregate I/O throughput for file reads scales linearly with respect to the increased number of threads. At four threads, it reaches 424.6MB/s. When a few of the OSTs are overloaded, we observe that there is a huge drop in throughput for the naive algorithm, for example, Naive(4,1) drops by 30% compared to the Naive(4,0) because it cannot avoid the congested OSTs. We can observe that there is only a 16% drop in throughput when using the layout-aware algorithm. For the layout-aware algorithm, we implement a heuristic algorithm for detecting congested servers, with three tunable parameters; window size (`W`), threshold (`T`), and the number of threads that skip the congested server (`M`). For the experiments in Figure 6, we used 40 for the window size (`W`), 0.05sec for the threshold (`T`), based on the latency of 1MB I/O operation on an OST, and 16 for `M`. Figure 6(b) shows the impact of varying the window size to the overall performance of the layout-aware algorithm. We observe that the throughput slightly improves as the window size increases, however, it significantly drops down to 220-230MB/s when `W` reaches 1,000. It implies that

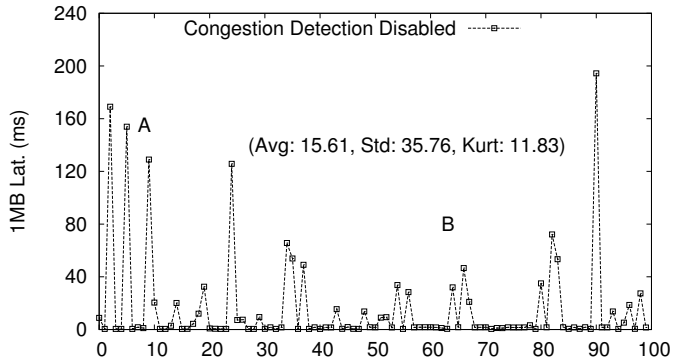


(a) Source Algorithm

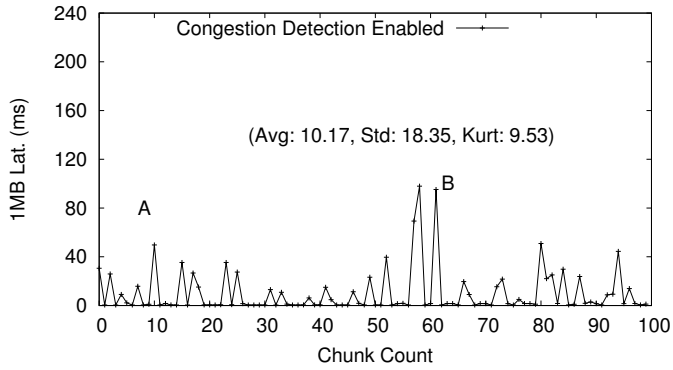


(b) Sensitivity Test

Figure 6. Performance comparison for source algorithms. In (x, y), 'x' indicates the number of threads operating for file reads. 'y' indicates the status of congestion on the storage systems. '1' indicates that a few of OSTs are overloaded, and '0' means it's not.



(a) Congestion Detection Engine Disabled



(b) Congestion Detection Engine Enabled

Figure 7. Time-series analysis of layout-aware algorithm when the congestion detection engine is disabled, and enabled. The average, standard deviation, and kurtosis are shown on each plot. Kurtosis characterizes the relative peakedness compared with the normal distribution.

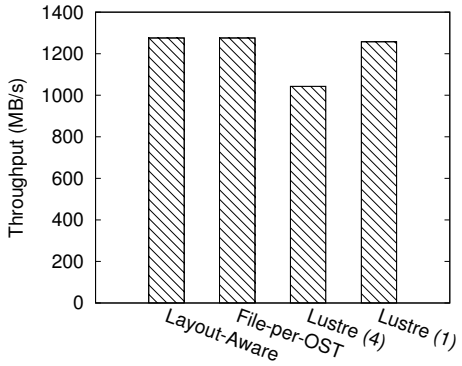
the window size needs to be carefully set in the algorithm, otherwise, it can reduce performance.

Figure 7 shows the efficacy of the congestion detection engine with the layout-aware algorithm in the analysis of the time series. We measured the response time of every 1MB read. The upper plot in Figure 7 shows the results when the congestion detection engine is disabled, and the lower plot shows when it is enabled. The congested OST can cause an increase in service time for the request. This in turn causes the pending request in the I/O driver queue to incur longer latency. For example, a request in the region marked 'A' on the plot competes with another workload on the OST, resulting in very high I/O service time. In sharp contrast, during the same period, the layout-aware algorithm with the congestion detection engine enabled is able to keep the I/O service time low and provide sustained improved performance. However, we also see a slight increase in service time in the region marked 'B', most likely due to an incorrect decision by the congestion detection engine.

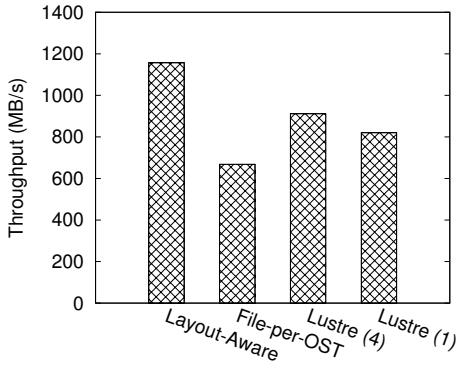
Figure 8 compares the results of sink algorithms for W(sink) in Table I. When none of the OSTs are overloaded, we see that Lustre(4) shows a lower throughput than Lustre(1). With Lustre(1), the I/O streams access separate servers so that there is less interference between them. Overall, all four cases show higher throughputs between 1-1.2GB/s than those in Figure 6. Note that we ran with eight threads for W(sink). When a few of the OSTs are overloaded, all cases except for the layout-aware algorithm experience reduced bandwidth by an average of 37% from the max throughput (1.27GB/s) when not congested. The layout-aware algorithm, in contrast, experiences a drop of just under 10% from the max throughput achieved when not congested.

VI. Summary and Future Work

As data-sharing in scientific data sets is growing, data movement becomes a critical component helping building virtual computing and data centers by computing geographically distributed facilities. In this paper, we address the issue of competing workloads accessing a parallel file system becoming a bottleneck for bulk data movement in and out of the PFS. We investigated the I/O load imbalance problems using actual performance data collected from the Spider storage system. We also examined several I/O scheduling



(a) Normal condition



(b) Congested condition

Figure 8. Performance comparison for sink algorithms. All experiments were done using eight threads. The number in the parentheses denotes stripe count for Lustre.

algorithms to optimize I/O performance at data source and sink hosts and we showed that local scheduling policies with layout-awareness can help match impedance within a single host. Initial results are promising and we need to further examine algorithms for large scale resources. For future work, we plan to implement the scheduling of parallel file system I/O operations in concert with network transfers using CCI and to investigate hierarchical storage awareness using non-volatile memory devices in the orchestration framework.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their detailed comments which helped us improve the quality of this paper This research is sponsored by the Office of Advanced Scientific Computing Research; U.S. Department of Energy and used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at Oak Ridge National Laboratory, which is supported by the Office of Science of the Department of Energy under Contract DE-AC05-00OR22725.

REFERENCES

[1] William Allcock, John Bresnahan, Rajkumar Kettimuthu, Michael Link, Catalin Dumitrescu, Ioan Raicu, and Ian Foster. The Globus Striped GridFTP Framework and Server. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, Washington, DC, USA, 2005. IEEE Computer Society.

[2] Scott Atchley Atchley, David A. Dillow, Galen M. Shipman, Patrick Geoffray, Jeffrey M. Squyres, and Georgie Bosilca. The Common Communication Interface (CCI). In *Hot Interconnects*, pages 51–60, 2011.

[3] Andrew Hanushevsky. BBCP. http://www.slac.stanford.edu/~abh/bbcp/.

[4] Youngjae Kim, Raghul Gunasekaran, Galen M. Shipman, and David A. Dillow. Workload Characterization of a Leadership Class Storage. In *Proceedings of 2010 5th Petascale Data Storage Workshop*, PDSW '10, pages 1–5, 2010.

[5] Qing Liu, Norbert Podhorszki, Jeremy Logan, and Scott Klasky. Runtime I/O Re-Routing + Throttling on HPC Storage. In *Proceedings of the Workshop on Hot Topics in Storage and File Systems*, HotStorage '13, 2013.

[6] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing Variability in the IO Performance of Petascale Storage Systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–12, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Ningfang Mi, Alma Riska, Qi Zhang, Evgenia Smirni, and Erik Riedel. Efficient Management of Idleness in Storage Systems. *Trans. Storage*, 5(2):4:1–4:25, June 2009.

[8] Ross Miller, Jason Hill, David A. Dillow, Raghul Gunasekaran, Galen M. Shipman, and Don Maxwell. Monitoring Tools For Large Scale Systems. In *Proceedings of the Cray User Group Meeting*, CUG '10, 2010.

[9] Oak Ridge National Laboratory. I/O Benchmark Suite. https://www.olcf.ornl.gov/center-projects/file-system-projects/.

[10] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, Thomas Robertazzi, Brian L. Tierney, and Eric Pouyoul. Protocols for Wide-Area Data-Intensive Applications: Design and Performance Issues. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 34:1–34:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[11] Bradley Settlemyer, Jonathan M. Dobson, Stephen W. Hodson, Jeffery A. Kuehn, Stephen W. Poole, and Thomas M. Ruwart. A Technique for Moving Large Data Sets over High-Performance Long Distance Networks. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society.

[12] Galen M. Shipman, David A. Dillow, Sarp Oral, Feiyi Wang, Douglas Fuller, Jason Hill, and Zhe Zhang. Lessons Learned in Deploying the World's Largest Scale Lustre File System. In *Proceedings of the Cray User Group Meeting*, CUG '10, 2010.