# LADS: Optimizing Data Transfers Using Layout-Aware Data Scheduling

Youngjae Kim, Scott Atchley, Geoffroy R. Vallée, and Galen M. Shipman, *Oak Ridge National Laboratory*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/kim

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

# LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling

Youngjae Kim, Scott Atchley, Geoffroy R. Vallée, Galen M. Shipman

*Oak Ridge National Laboratory*
*{kimy1, atchleyes, valleegr, gshipman}@ornl.gov*

## Abstract

While future terabit networks hold the promise of significantly improving big-data motion among geographically distributed data centers, significant challenges must be overcome even on today's 100 gigabit networks to realize end-to-end performance. Multiple bottlenecks exist along the end-to-end path from source to sink. Data storage infrastructure at both the source and sink and its interplay with the wide-area network are increasingly the bottleneck to achieving high performance. In this paper, we identify the issues that lead to congestion on the path of an end-to-end data transfer in the terabit network environment, and we present a new bulk data movement framework called *LADS* for terabit networks. *LADS* exploits the underlying storage layout at each endpoint to maximize throughput without negatively impacting the performance of shared storage resources for other users. *LADS* also uses the Common Communication Interface (CCI) in lieu of the sockets interface to use zero-copy, OS-bypass hardware when available. It can further improve data transfer performance under congestion on the end systems using buffering at the source using flash storage. With our evaluations, we show that *LADS* can avoid congested storage elements within the shared storage resource, improving I/O bandwidth, and data transfer rates across the high speed networks.

## 1 Introduction

While "Big Data" is now in vogue, many DOE science facilities have produced a vast amount of experimental and simulation data for many years. Several U.S. Department of Energy (DOE) leadership-computing facilities, such as the Oak Ridge Leadership Computing Facility (OLCF) [23], the Argonne Leadership Computing Facility (ALCF) [1], and the National Energy Research Scientific Computing (NERSC) [21] generate hundreds of petabytes per year of simulation data and are projected to generate in excess of 1 exabyte per year by 2018 [31]. The Big Data and Scientific Discovery report from the DOE, Office of Science, Office of Advanced Scientific Computing Research (ASCR) [5], predicts one of scientific data challenges is the worsening input/output (I/O) bottleneck and the high data movement cost.

To accommodate growing volumes of data, organizations will continue to deploy larger, well provisioned storage infrastructures. These data sets, however, do not exist in isolation. For example, scientists and their collaborators who use the DOE's computational facilities typically have access to additional resources at multiple facilities and/or universities. They use these resources to analyze data generated from experimental facilities or simulation on supercomputers and to validate their results, both of which requires moving the data between geographically dispersed organizations. Some examples of large collaborations include: 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 order to support the increased growth of data and the desire to move it between organizations, network operators are increasing the capabilities of the network. DOE's Energy Sciences Network (ESnet) [32], for example, has upgraded its network to 100 Gb/s between many DOE facilities, and future deployments will most likely support 400 Gb/s followed by 1 Tb/s throughput. However, these network improvements only contribute to improving the network data transfer rate, not end-to-end data transfer rate from source storage system to sink storage system. The data transfer nodes (DTN) connected to these storage systems and the wide-area network are the focal point for the impedance match between the faster networks and the relatively slower storage systems. In order to improve the scalability, parallel file systems (PFS) use separate servers to service metadata and I/O operations in parallel. To improve I/O throughput, the PFS use ever higher counts of I/O servers connected more disks. DOE sites have widely adopted various PFS to support both high performance I/O and large data sets. Typically, these large scale storage systems use tens to hundreds of I/O servers, each with tens to hundreds of disks, to improve scalability of performance and capacity.

Even as networks reach terabit speeds and PFS grow to exabytes, the storage-to-network mismatch will likely continue to be a major challenge. More importantly, such storage systems are shared resources servicing multiple clients including large computational systems. As contention for these large resources grows, there can be serious Quality-of-Service (QoS) differences between the observed I/O performance by users [11, 38]. Moreover, disk services can degrade while disks in the redundant array of independent disks (RAID) are rebuilding due to failed disks [37, 13]. Also, I/O load imbalance is a serious problem in parallel storage systems [18, 20]. The results showed that a few controllers are highly overloaded while most are not. These observations strongly motivate us to develop a mechanism to avoid temporarily congested servers during data transfers.

We investigate the issues related to designing a data transfer protocol using Common Communication Interface (CCI) [3, 33], that can fully exploit zero-copy, operating system (OS) bypass hardware when available and fall back to sockets when it is not. In particular, we focus on optimizing an end-to-end data transfer, and investigate the interaction between applications, network protocols, and storage systems at both source and sink hosts. We address various design issues for implementing data transfer protocols such as buffer and queue management, synchronization between worker threads, parallelization of remote memory access (RMA) transfers, and I/O optimizations on storage systems. With these design considerations, we develop a **L**ayout-**A**ware **D**ata **S**cheduler (*LADS*).

In this paper, we present *LADS*, a bulk data movement framework for use between PFS which uses the CCI interface for communication. Our primary contribution is that *LADS* uses the *physical view of files*, instead of a logical view. Traditional file transfer tools employ a logical view of files, regardless of how the underlying objects are distributed within the PFS. *LADS*, on the other hand, understands the physical layout of files in which (i) files are composed of data objects, (ii) the set of storage targets that hold the objects, and (iii) the topology of the storage servers and targets.[1] *LADS* aligns all reads and writes to the underlying object size within the PFS. Moreover, *LADS* allows out-of-order object transfers.

Our focus on the objects, rather than on the files, allows us to implement layout-aware I/O scheduling algorithms. With this, we can minimize the stalled I/O times due to congested storage targets by avoiding the congested servers and focusing on idle servers. All other existing data transfer tools [12, 2, 29, 27, 30] implicitly synchronize per file and focus exclusively on the servers that store that one file whether they are busy or not. We also propose a congestion-aware I/O scheduling algorithm, which can increase the data processing rate per thread, leading to a higher data transfer rate. We also implement and evaluate the ideas of hierarchical data transfer using non-volatile memory (NVM) devices. Especially, in an environment where I/O loads on storage dynamically vary, there can be a slow storage target due to congestion.

We conduct a comprehensive evaluation for our proposed ideas using a file size distribution based on a snapshot of one of the file systems of Spider (the previous file system) at ORNL. We compare the performance of our framework with a widely used data transfer program, bbcp [12]. Specifically, in our evaluation with the real file distribution based workload, we observe that our framework yields a 4-5 times higher data transfer rate than bbcp when using eight threads on a node. Also, we find that with a small amount of SSD, *LADS* can improve further the data transfer rate by 37% over a baseline without SSD buffering and far more cost-effectively than provisioning additional DRAM.

## 2 Background

We first introduce our target environment for DOE data movement frameworks - the data life cycle, network environment, and data storage infrastructure. Next, we define I/O optimization problems at a multi-level hierarchy in the PFS.

### 2.1 Target Environment

DOE has large HPC systems (e.g. OLCF's Titan and ALCF's Mira) and scientific instruments (e.g. ORNL's SNS and ANL's APS) which generate large, bulk, synchronous I/O. The HPC systems run simulations that have intense computational phases, followed by interprocess communication, and periodically by I/O to checkpoint state or save an output file. The simulation's startup is dominated by a read phase to retrieve the input files as well as the application binary and libraries. The instruments, on the other hand, do not have a read phase and strictly have write workloads that capture measurements. These measurements are triggered by a periodic event such as an accelerated particle hitting a target which generates various energies and sub-particles. The instrument's detectors will capture these events and it must move the data off the device before the next event.

In order to store this data, these systems typically have large PFS connected by a high-performance network.

[1] We use Lustre terminology for object storage servers (OSS) and targets (OST). An OST manages a single device. A single Lustre OSS manages one or more OSTs.

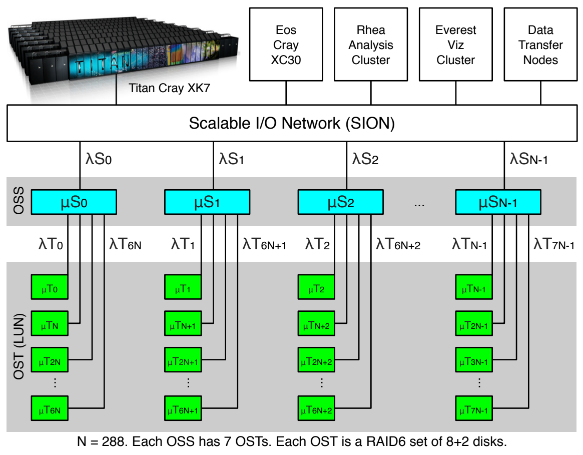


Figure 1: OLCF center-wide PFS and clients

Some sites, such as OLCF, have a center-wide file system accessible by multiple HPC systems as well as by analysis and visualization clusters. In this case, the HPC systems are the primary users and the clusters are secondary users. Given the cost to run these larger resources, one would not want to negatively impact the HPC system's performance due to I/O by the secondary users.

Lastly, most of the scientists using these systems are not located within the facility. Most are from other DOE sites, universities, as well as some commercial entities. They eventually want to move the data back to their institutions in order to further analyze the data. Each site has several data transfer nodes (DTN) that mount the PFS and are connected to DOE's Energy Sciences Network (ESnet). ESnet currently provides 100 Gb/s connectivity to over 40 DOE institutions as well as peering with Internet2 and commercial backbone providers. The DTNs currently have 10 Gb/s NICs but will migrate to 40 Gb/s NICs in the near future. As with the analysis and visualization clusters, use of the DTNs should not negatively impact the I/O of the large HPC systems.

## 2.2 Spider Storage Systems for Titan

Spider II is OLCF's second generation, center-wide, Lustre system. Its primary client is Titan [25, 19], currently ranked second in the Top500. Titan has 18,688 compute nodes, which mount Spider directly. Titan's I/O traffic passes through 432 I/O nodes, which act as Lustre Networking (LNET) routers between Titan's Cray Gemini network and OLCF's InfiniBand$^{\text{TM}}$(IB) Scalable I/O Network (SION). In addition to Titan, Spider is shared with Eos, a 744 node Cray XC30 system, analysis and visualization clusters, and DTNs. Figure 1 provides an overview of Spider. Currently the file system is accessible via two name-spaces, *atlas1* and *atlas2*, for load-balancing and capacity management purposes. Each namespace has 144 OSSes, which manage seven OSTs each, for a total of 1,008 OSTs per namespace. Each OST represents a RAID-6 set of ten (8+2) disks.

## 2.3 Problem Definition: I/O Optimization

**I/O Contention and Mitigation:** A storage server experiences transient congestion when competing I/O requests exceed the capabilities of that server. During these periods, the time to service each new request increases. This is a common occurrence within a PFS when either a large application enters its I/O phase (e.g. writing a checkpoint, reading shared libraries on startup) or multiple applications are accessing files co-located on a subset of OSTs. Disk rebuild processes of a RAID array can also delay I/O services. OS caching and application-level buffering can sometimes mask the congestion for many applications, but data movement tools do not benefit from these techniques. If the congestion occurs on the source side of the transfer, the source's network buffers will drain and eventually stall. On the other hand, congestion at the sink will cause the buffers of both the sink and then the source to fill, eventually stalling the I/O threads at the source. We refer to threads stalled on I/O accesses to congested OSTs as *stalled I/Os*. We try to lower the storage occupancy rate of stalled I/Os in order to minimize the impact of storage congestion on the overall I/O performance using three techniques: *Layout-aware I/O Scheduling*, *OST congestion-aware I/O scheduling*, and *object caching on SSDs*.

**Two-level bottlenecks:** Figure 1 illustrates the potential places for I/O bottlenecks when accessing OSTs via OSSes in Lustre file systems. For $OSS_m$, if the arrival rate ($\lambda_{OSSm}$) is greater than its service rate ($\mu_{OSSm}$), the server will start to overflow, becoming the bottleneck and its incoming service will be delayed. This can happen if the number of OSTs connected to an OSS is greater than what the network connection to the OSS can handle. To avoid this case, OLCF provisions the number of OSTs per OSS such that $\mu_{OSSm} > \sum_{j=1}^{k} \mu_{OSSn+j}$. Even if $\lambda_{OSSm}$ is smaller than $\mu_{OSSm}$, OSTs can become the bottleneck. For example, if $\lambda_{OST\,j}$ is greater than $\mu_{OST\,j}$, $OST_j$ becomes the bottleneck. Therefore, *LADS* has to avoid both server and target bottlenecks in a way that it does not assign I/O threads to the overloaded server or target.

**Lustre Configuration Impacts I/O Contention:** In Lustre, a file's data is stored in a set of objects. The underlying transfers are 1 MB aligned on 1 MB boundaries. If the stripe count is four, then the first object holds offsets 0, 4 MB, 8 MB, etc. Each object is stored on a separate OST. The mapping of the OSTs to the OSSes can impact how a file's object are stored. Figure 2 shows how the OST-to-OSS mapping can physically impact a file's object placement. The default mapping is to assign OSTs sequentially to OSSes. For a file with a stripe count of

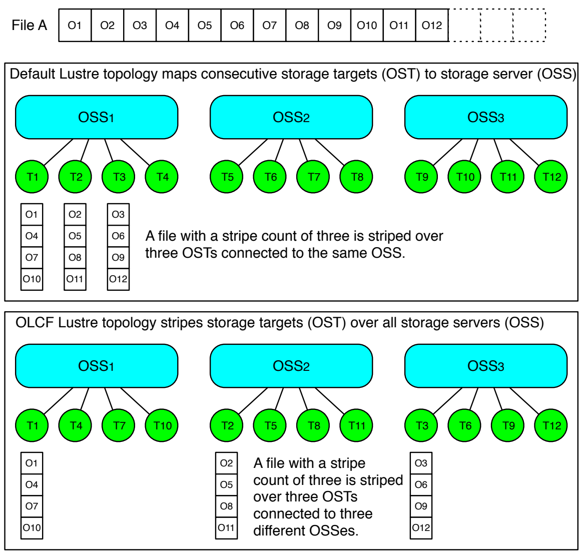

File A
O1 O2 O3 O4 O5 O6 O7 O8 O9 O10 O11 O12
Default Lustre topology maps consecutive storage targets (OST) to storage server (OSS)
OSS1 OSS2 OSS3
T1 T2 T3 T4 T5 T6 T7 T8 T9 T10 T11 T12
A file with a stripe count of three is striped over three OSTs connected to the same OSS.
OLCF Lustre topology stripes storage targets (OST) over all storage servers (OSS)
T1 T4 T7 T10 T2 T5 T8 T11 T3 T6 T9 T12
A file with a stripe count of three is striped over three OSTs connected to three different OSSes.

Figure 2: File striping in Lustre.

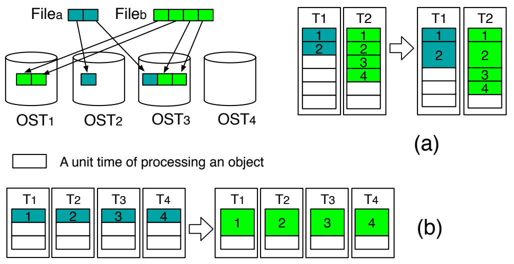


Figure 3: Illustration of slow-down of each job(T) due to OST contention when accessing the same resource at the same time.

three and four OSTs per OSS, the objects will be stored on three OSTs connected to one OSS. OLCF, on the other hand, uses a mapping such that OSTs are assigned round-robin over all the OSSes. In this example, a file with a stripe count of three is assigned to three OSTs and each OST is connected to a separate OSS.

Depending on the choices of storage and networking hardware, the OSS or the OSTs may be the bottleneck. To improve the I/O throughput by minimizing contention, the higher layers need this information.

**Logical versus Physical File View:** Traditional file transfer tools [12, 2] rely on the logical view of files (called *File-based approach*), which ignores the underlying file system architecture. An I/O thread can be assigned to a complete file, and it should work on the file until the entire file is read or written. If more than one thread is used, these threads might compete for the same OSS or OST, causing server or disk contention respectively. Such contention can result in the slow-down of applications.

To demonstrate how the File-based approach, which is unaware of the underlying file system layout, contributes to the problem of I/O contention in the PFS, we use a simple example in Figure 3, in which we assume each OST can service an object at a time within a fixed service time. In the figure, $File_a$ is striped over $OST_2$ and $OST_3$ and $File_b$ is striped over $OST_1$ and $OST_3$. In Figure 3(a), Thread 1 ($T_1$) and $T_2$ attempt to read $File_a$ and $File_b$ at the same time respectively. $T_1$ and $T_2$ read different files, however, $T_1$ and $T_2$ can interfere each other on accessing the same OST. Based on a dilation factor model [17], as $T_1$ and $T_2$ complete $OST_3$ , $T_1$ and $T_2$ can slow down by 25% and 12.5% respectively. In Figure 3(a), all four threads access different logical regions of the same $File_b$, however, as $T_1$ and $T_2$ complete for $OST_1$, and $T_3$ and $T_4$ complete for $OST_3$. Thus, each thread slows down by 50%. The results of this example indicates that OST contention may increase due the lack of understanding of the physical layout of the file's objects.

In contrast, *LADS*, views the entire workload from a physical point of view based on the underlying file system architecture. *LADS* consider the entire workload of $O$ objects, where $O$ is all of the objects in the $N$ total files, and each object represents one transfer MTU of data. It can also exploit the underlying storage architecture, and can use the file layout information for scheduling accesses of OSTs. Thus it takes into account the $S$ servers and $T$ targets that hold the $O$ objects. We then load-balance based on the physical distribution of the objects. A thread can be assigned to an object of any file on any OST without requiring that all objects of a particular file be transferred before objects of another file.

## 3 Design of LADS

*LADS* is motivated to answer a simple question: *how can we exploit the underlying storage architecture to minimize I/O contention at the data source and sink?* In this section, we describe our design rational behind the *LADS* implementation, system architecture, and several key design techniques using a physical view of files on the underlying file system architecture.

We have implemented a data transfer framework with the following main design goals – (i) improved parallelism, (ii) network portability, and (iii) congestion-aware scheduling. Our design tries to maximize parallelism by overlapping as many operations as possible, using use a combined threading and an event-driven model.

### 3.1 LADS Overview

**System Architecture:** Figure 4 provides an overview of our design and implementation for I/O sourcing and sinking for a PFS. *LADS* is composed of the follwoing threads: The *Master* thread maintains transfer state, while *I/O* threads read and write objects of files from and to the PFS. The *Comm* thread is in charge of all data transfers between source and sink. In our implementation, there is one Master thread, a configurable num-

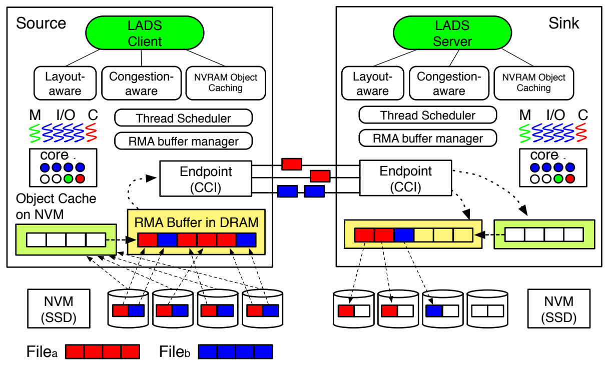


Figure 4: An architecture overview.

ber of I/O threads, and one Comm thread. Because the I/O threads use blocking calls, we allow more threads than cores (i.e. over-subscription). Since we can over-subscribe the cores, the Master and I/O threads block when idle or waiting for a resource. The Comm thread never blocks and always tries to progress communication. The Comm thread generates most of the events that drive the application. If the Comm thread needs a resource (e.g. a buffer) which it cannot get immediately, it will queue the request on the Master's queue and wake the Master.

Several I/O optimization techniques are implemented in *LADS*. In the figure, the layout-aware technique can optimize the unit size of the data accessed by the I/O threads to object size in the underlying file systems, and improved the stalled I/O time when the server is congested. The OST congestion-aware algorithm can avoid the congested servers. NVM can be used as an extended memory region, when the RMA buffer full using the object caching technique. The Comm threads at source and sink, using CCI, pin memory regions for RMA transfers between the Comm threads at source and sink. At the source, if the RMA buffer is full, the Master will notify I/O threads to use the NVM buffer instead of directly copying objects from the PFS into the RMA buffer, thus, it allows pre-loading on the extended memory regions on the NVM. At sink, if the RMA buffer is full, likewise, the extended NVM regions can be used as an intermediate buffer before the PFS to avoid stalling the network transfers.

## 3.2 Data Structure Overview

We organized the various data structures to minimize false sharing by the various threads. The global state includes a lock which is used to synchronize the threads at startup and shutdown as well as to manage the number of files opened and completed. Other locks are resource specific. There are two wait queues, one for the Master and the other for the I/O threads. When using the SSD to provide additional buffering, it has a wait queue as well.

The Master and I/O thread structures also have a waiter structure that includes their condition variable and an entry for the wait queue. The Master and Comm threads have a work queue implemented using a doubly-linked list. The I/O threads will pull requests off of the OST work queues (described below).

We manage the open files using the GNU tree search interface, which is implemented as a red-black tree. The tree has its own mutex and counter. We manage the RMA and SSD buffers using bitmaps that indicate which offsets are available (the offset is the index in the bitmap multiplied by the object size), an array of contexts (used to store block requests using that buffer), and a mutex. Lastly, because our implementation currently targets Lustre, we have an array of OST pointers. Each OST has a work queue, mutex, queue count, and busy flag. The number of OST queues is determined by the number of OSTs in the PFS. The design can easily be extended to other PFS.

To avoid threads spinning on mutexes as well as "thundering herds" [14] when trying to acquire a resource, we use per-resource wait queues consisting of a linked list, a mutex, and a per-thread condition variable. If the resource is not available, the waiter will acquire the lock, enqueue itself on the resource's wait list, and then block on its own condition variable. When another user wants to release the resource, it acquires the lock, dequeues the first waiter, releases the lock, and signals the waiter's condition variable. This ensures fairness and avoids spinning and thundering herds.

## 3.3 Object Transfer Protocol

For transferring files, first the source and sink processes (hereafter simply source and sink) need to initialize some state, spawn threads, and exchange some information. The initial state includes a global lock used to synchronize at startup, the various wait queues, the file tree to manage open files, the OST work queues, and the structure for managing access to the RMA buffer. The Master thread initializes its work queue, its wait queue, and the wait queue for I/O threads.

The Comm thread opens a CCI endpoint (send and receive queues, completion queue), allocates its RMA buffer and registers it with CCI, and opens a connection to the remote peer. The source Comm thread sends its maximum object size, number of objects in the RMA buffer, and the memory handle for the RMA buffer. The sink Comm thread accepts the connection request, which triggers the CCI connect event on the source. The I/O threads simply wait for the other threads.

After the CCI initialization step, data transfer will follow these steps at source and sink. The detail figures of thread communication between source and sink hosts can be found in our technical report [16].

***Step 1.*** For each file, (i) the source's master will open the file, determine the file's length and layout (i.e. the size of the stored object and on which OSTs they are located), and generate a `NEW_FILE` request and enqueue that request on the Comm thread's work queue. (ii) The Comm thread generates `NEW_BLOCK` requests for each stored object and enqueue that request on the appropriate OSTs' work queues. (iii) The Comm thread will marshal the `NEW_FILE` request and send it to the sink.

***Step 2.*** At sink, the Comm thread will receive the `NEW_FILE` request and enqueue it on the Master's work queue and wake it up. The Master will open the file, add the file descriptor to the request, change the request type to `FILE_ID` and queue the request on the Comm's work queue. The Comm thread will dequeue it and send it to the source.

***Step 3.*** At source, when the Comm thread receives the `FILE_ID` message, it will wake up *N* I/O threads, where N is the number of OSTs over which the file is striped. An I/O thread first reserves a buffer registered with CCI for RMA. It then determines which OST queue it should access and then dequeues the first `NEW_BLOCK` request. It uses `pread()` to read the data into the RMA buffer. When the read completes, it enqueues the request on the Comm thread's work queue. The Comm thread marshals the request and sends it to the sink. Note, the source's Comm thread's work queue will have intermingled `NEW_FILE` and `NEW_BLOCK` requests thus overlapping file id exchange and block requests.

***Step 4.*** At sink, the Comm thread receives the request and attempts to reserve a RMA buffer. If successful, it initiates a RMA *Read* of the data. If not, it enqueues the request on the Master's work queue and wakes the Master. The Master will sleep on the RMA buffer's wait queue until a buffer is released. It then will queue the request on the Comm's queue, which will then issue the RMA Read.

***Step 5.*** At sink, when the RMA Read completes, it sends a `BLOCK_DONE` message back to the source. The sink's Comm thread determines the appropriate OST by the block's file offset and queues it on the OST's work queue. It then wakes an I/O thread. The I/O thread looks for the next OST to service, dequeues a request, calls `pwrite()` to write the data to disk. When the write completes, it releases the RMA buffer so the Comm thread can initiate another RMA Read.

***Step 6.*** When the source's Comm thread receives the `BLOCK_DONE` message, it releases the RMA buffer and wakes an I/O thread. This pattern continues until all of the file's blocks have been transferred. When all blocks have been written, the source sends a `FILE_DONE` message and closes the file. When the sink receives that message, it too closes the file.

## 3.4 Scheduling

**Layout-aware Scheduling:** In a PFS, the file is stored as a collection of objects and stored across multiple servers to improve overall I/O throughput. Best practices for accessing a PFS is for the application to issue large requests in order to reap the benefits of parallel accesses across many servers. A single thread accessing a file will request N objects and can read M objects (assuming $M < N$, and the file is striped over M servers) in parallel at once. If one of the servers is congested, however, the request duration is determined by the slowest server. So the throughput of the request for N objects is determined by the throughput of objects from the congested server. In contract, in our approach, instead of a single thread requesting N objects, we have N threads request one object each from separate servers, because we align all I/O accesses to object boundaries. If one of the requests is delayed by a congested server, the N-1 threads are free to issue new requests to other servers. By the time that the request to the slow server completes, we may be able to retrieve more than N objects.

While the aligned-access technique aims to reduce the I/O stall times and improve overall throughput, it does not specify to which servers to send requests. Most, if not all, data movement tools attempt to move one file at a time (e.g. bbcp, XDD) or a small subset (e.g. GridFTP) at a time. In a PFS, however, a single file is striped over N servers. In the case of the Atlas file system at ORNL, the default is four servers. Although the file system may have hundreds of storage servers, most data movement tools will access a very small subset of them at a time. If one of those servers is congested, overall performance will suffer during the congested period.

**Congestion-aware Scheduling:** For congestion-aware I/O scheduling, we attempt to avoid intermittently congested storage servers. Given a set of files, we determine where all of the objects reside in the case of reading at the source or determine which servers to stripe the objects over when writing at the sink. We then schedule the accesses based the location of the objects, not based on the file. We enqueue a request for a specific object on a particular OST's queue. The I/O threads then select a queue in a round-robin fashion and dequeue the first request. If another thread is accessing an OST, the other threads skip that queue and move on to the next. If one OST is congested, a thread may stall, but the other threads are free to move on to other, non-congested servers. This is important in a HPC facility like ORNL. The PFS's primary user is the HPC system. We do not want to tune the data movement tools such that they reduce the performance of the HPC system, which is a very expensive resource. Our goal is to maximize performance while using the lightest touch on the PFS.

The basic per-OST queues and simple round-robin scheduling over all the OSTs is able to improve overall I/O performance. We then extend layout-aware scheduling to be congestion-aware by implementing a heuristic algorithm to detect and avoid the congested OSTs. The algorithm can make proactive decisions for selecting storage targets that next I/O threads will work on. The algorithm uses a threshold-based throttling mechanism to further lessen our impact on the HPC system's use of the PFS. When reading at the source, for example, an I/O thread reads a object from its appropriate server and records the read time, and computes an average of multiple object read times during a pre-set time window time ($W$). If the average read time during $W$ is greater than the pre-set threshold value ($T$), then it marks the server as congested. The algorithm tells the threads that they should skip congested servers $M$ times. Consequently, the I/O threads avoid the congested servers for a short amount of time, leading to the reduced I/O stall times.

### 3.5 Object Caching on SSDs

In the case when the sink is experiencing wide-spread congestion (i.e. every I/O thread is accessing a congested server), newly arriving objects will quickly fill the RMA buffer. The sink will then stall the pipeline of RMA Read requests from the source causing the source's RMA buffer to fill. Once full, the source's I/O threads will stall because they have no buffers in which to read. To mitigate this, we investigate using a fast NVM device to extend the buffer space available for reading at the source. Several efforts have introduced new interfaces to efficiently use NVM as an extended memory region [4, 35, 9, 24]. In this work, we specifically use the *NVMalloc* library [35] to build a NVM based, intermediate buffer pool at the source using fast PCIe-based COTS SSDs, where we create a log-file memory-mapped using a `mmap()` system call. The key use of NVM buffer pool is to continue reading objects when the RMA buffer is full at source.

In our implementation, when servicing a new request, an I/O thread tries to reserve a RMA buffer. If one is not available, it attempts to reserve one in the SSD buffer. If successful, it reads into the SSD buffer, enqueues the request on a SSD queue, and wakes the SSD thread. The SSD thread then attempts to acquire a RMA buffer. If not available, it sleeps waiting for a RMA buffer to be released. When a buffer is released, it wakes, reserves the RMA buffer, copies the data to the RMA buffer, and enqueues the request on the Comm thread's work queue. Lastly, the Comm thread marshals the `NEW_BLOCK` and sends it off to the sink.

We could apply the same idea of source-side SSD buffering algorithm for sink-side SSD buffering, however, as we will discuss in the evaluation section in detail, sink-side buffering does little to improve data transfer rates, when buffered I/Os are allowed. Typically writes are buffered I/Os. The key for the SSD buffering is to decide when to use the SSD buffer or not. When using buffered I/Os at sink, our algorithm can not account for the effect of OS's buffer cache and fails to correctly detect congested servers. Using direct I/O for the writes is possible and would allow our algorithm to detect congested servers, but direct I/O performs much worse and we chose not to use it for sink-side SSD buffering.

The copy from SSD buffer to RMA buffer is needed when using hardware that supports zero-copy RMA because the memory must be pinned and registered with the hardware and we cannot register the mapped SSD file. Our design does this even when the hardware does not provide RMA support (i.e. when using sockets underneath CCI). We could detect this scenario and avoid the copy by sending directly from the SSD buffer, but we do not implement at this time. Also, should future interconnects support RMA from NVM, we could avoid the copy as well.

## 4 Evaluation

For the evaluation of *LADS*, we use two experimental environments, without and with server congestion, and our production environment. First, we show the results of *LADS* without congested servers. We explore the effectiveness of object scheduling in *LADS* versus file based scheduling (e.g., bbcp). We then explore the performance of *LADS* by varying RMA buffer size and scaling performance. Second, we study the impact of server congestion on *LADS* and we propose two mitigating strategies, congestion-aware I/O scheduling and SSD buffering. Lastly, we show the DTN to DTN evaluation with ORNL production systems.

### 4.1 Experimental Systems

**Implementation:** *LADS* has been implemented using 4 K lines of C code using Pthreads. We used CCI, which is an open-source network abstraction layer, downloadable from CCI-Forum [6]. The communication model follows a client-server model. On the server side, the *LADS* server daemon has to be run before the *LADS* client starts to transfer data.

**Test-bed:** In this setup, we used a private testbed with two nodes (source and sink) connected by InfiniBand (IB) QDR (40 Gb/s). The nodes used the IB network to communicate with each other and the disk arrays. We used two Intel® Xeon® CPU E5-2609 @ 2.40 GHz servers with eight cores, 256 GB DRAM, and two node-local Fusion-io Duo SSDs [10] for data transfer nodes (source and sink hosts) running with Linux kernel 2.6.32-358.23.2. Both the source and sink nodes have separate Lustre file systems with one OSS server, one MDS server, and 32 OSTs, mounted over 32 SAS 10K

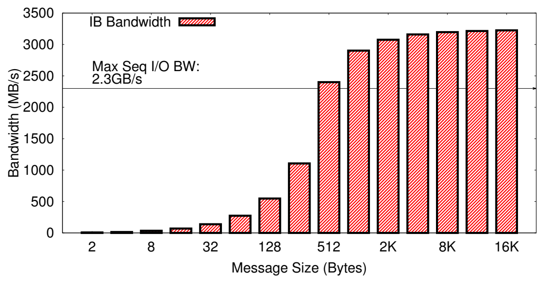


Figure 5: IB vs. storage bandwidth.

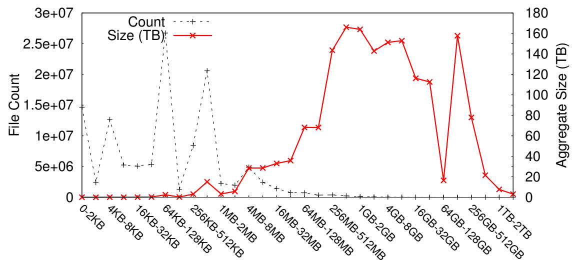


Figure 6: File size distribution.

RPM 1TB drives each. For each file systems, we created 32 logical volume drives on top of the drives to have each disk to become an OST.

To fairly evaluate our implementation framework, we ensured that storage server bandwidth is not over-provisioned with respect to network bandwidth between those source and sink servers (i.e., the network would not be the bottleneck). Figure 5 shows the results on comparing network and storage I/O bandwidths in our test-bed. For storage bandwidth, we ran block level I/O benchmarks [22] on a host to 32 disk volumes in parallel with 1 MB sequential I/O streams on each benchmark with the highest queue depth of 16. The IB bandwidth increases as the message size increases, and it reaches about 3.2 GB/s, whereas the I/O bandwidth is measured around 2.3 GB/s at the most. This testbed allowed us to replicate the temporal congestion of the disks to provide fair comparisons between *LADS* and bbcp.

**Production system:** We have also tested *LADS* and bbcp between our production Data Transfer Nodes (DTNs), connected to two separate Lustre file systems at ORNL. Each DTN is connected to the OLCF backbone network via a QDR or FDR IB connection to the OLCF's Scalable I/O Network where Atlas' Lustre file systems are mounted (Refer to Figure 1). In our evaluation, we measured the data transfer rate from atlas1 to atlas2 via DTN nodes with *LADS* and bbcp. In order to minimize the OS page-cache effect, we cleared out OS page cache before each measurement at both test-bed and production system.

**Workloads:** For a realistic performance comparison, we used a file system snapshot taken for a `widow3` partition in the Spider-I file systems hosted by ORNL in 2013 to determine file set sizes. Figure 6 plots a file size distribution in terms of the number of files and the aggregate size of files. We can observe that 90.35% of the files are less than 4 MB and 86.76% are less than 1 MB. Less than 10% of the files are greater than 4 MB. On the other hand, the larger files occupy most of the file system space. For the purpose of our evaluation, we used two representative file sizes to have two file groups; one for *small files* with 10,000 1MB files, and the other for *big files* with 100 1GB files.

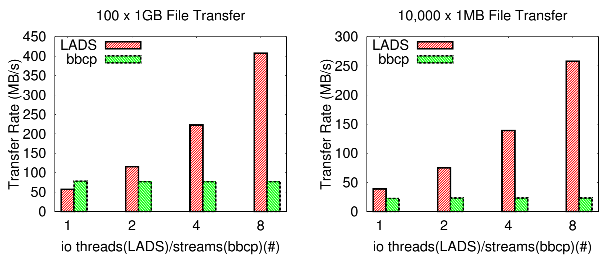
## 4.2 Scheduling Objects versus Files in an Uncongested Environment

In this section, we show the effectiveness of object scheduling in *LADS* versus file-based scheduling used by bbcp in a controlled, uncongested environment. This section focuses only on the difference between object versus file scheduling; Sections 4.3 and 4.4 will examine two mitigation strategies for congested environments.

Within our controlled test-bed environment, we evaluate the performance of *LADS* for big and small data sets, and compare it against bbcp. In both sets, the stripe count is one (i.e. each file is stored in 1 MB objects on a single OST). We note that our tests with a higher file stripe count are shown in the results of production system in Section 4.5.

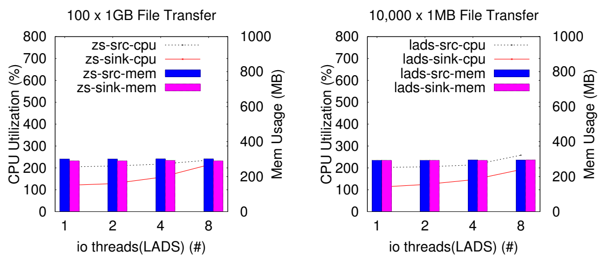
Figures 7 and 8 show the results of *LADS* and bbcp for these workloads. We had multiple runs for each test, however the variability was very small. Both experiments were tested while increasing the number of threads on each application. In *LADS*, we can vary the number of I/O threads, which can maximize CPU utilization on the data transfer node, but use a single Comm thread. On these hosts, *LADS* uses CCI's Verbs transport, which natively uses the underlying InfiniBand interconnect. In bbcp, we can only tune the number of TCP/IP streams for a performance improvement (bbcp always uses a single I/O thread). The streams ran over the same InfiniBand interconnect, but used the IPoIB interface which supports traditional sockets. Using Netperf, we measured IPoIB throughput at almost 1 GB/s. A newer OFED release should provide higher sockets performance, but we ensured that the network was never the bottleneck for these tests. In bbcp, we calculated the TCP window size ($W$) using the formula for bandwidth-delay product: using ping time ($T_{ping}$) and a network bandwidth ($B_{net}$) as follows: $W = T_{ping} \times B_{net}$. We used 10 MB for a TCP window size in our evaluation setup. We have also tested bbcp by varying the block size, however we have seen little performance difference between 1 MB and 4 MB, so we show the results with a block size of 1 MB for bbcp tests.

**Performance comparison for object scheduling of *LADS* and file-based scheduling of bbcp:** In Fig-



(a) Big Files (b) Small Files

Figure 7: Performance comparisons for *LADS* and bbcp. In bbcp, 10 MB is used for TCP window size.



(a) Big Files (b) Small Files

Figure 8: Resource utilization comparisons for *LADS*. CPU utilization is 800, when all cores are fully utilized.

ure 7(a)(b), we see that *LADS* shows almost a perfect linear scaling in terms of data transfer rate with respect to the increased number of I/O threads, whereas there is little improvement in bbcp with respect to the increased number of TCP/IP streams. bbcp is implemented using a file-based data transfer protocol in which, files are transferred one by one, and multiple TCP streams operate on the same file. Therefore, the bottleneck is determined by how wide the PFS stripes the file. We also found that with bbcp multiple TCP/IP streams will only offer a performance gain when a network speed is moderately slow compared with I/O bandwidth of the storage. Overall, we observe that *LADS* significantly outperforms bbcp for all test cases in Figure 7, except for the results when *LADS* transfer uses one I/O thread for a big file set. In this case, we believe that bbcp is benefiting from hardware-level read-ahead in our testbed. *LADS* did not benefit from it because the round-robin access of the I/O queues might mean that we are accessing an object from a different file the next time we visit this OST and lose the benefit of read-ahead. OLCF production systems disable read-ahead for this reason.

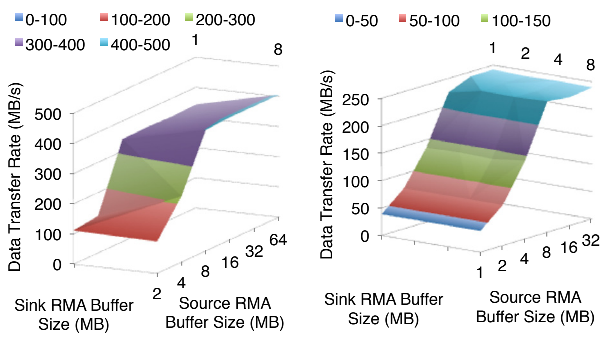
In *LADS*, we observe the maximum throughput at around 400-450MB/s for the experiment of a big data set, which is reasonable based on our test-bed configuration. The block-level throughput for all 16 disks is 2.3GB/s, the file system overhead reduces that by about 40% to 1.3-1.4GB/s. We tested with up to eight threads reducing the optimum to 650-700MB/s. Given thread synchronization overhead, 400-500MB/s is reasonable but improvement is still possible.

**Resource utilization in *LADS*:** *LADS* uses `DIRECT IO` for the source's read operations to minimize the resource utilization for CPU and memory, while the sink writes using buffered I/O. As we see from Figure 8(a)(b), *LADS* moderately uses system resources, and there is only a slight increase in CPU utilization as the number of I/O threads increases. The more I/O threads involve the more meta data service requests to Lustre file system and more I/O. Overall, *LADS* take advantage of the InfiniBand NIC's offloading abilities and manages well the system resources; CPU utilization stays relatively low even at eight I/O threads. Memory usage never varies and is almost constant at 280-300 MB. We used 256 MB for RMA buffer at source and sink, which accounts for the majority of memory usage.

On the contrary, in bbcp, we observe that CPU and memory usages are very low. For example, for small file workloads, when eight streams are used, their memory usage and CPU utilization are less than 2 MB and 5%, respectively. For big file workloads, with the same configuration, bbcp's memory usage and CPU utilization are at most about 30MB and less than 40%, respectively. Not surprisingly with bbcp's file-based approach, disk I/Os are the bottleneck so that the host resources cannot be fully utilized.

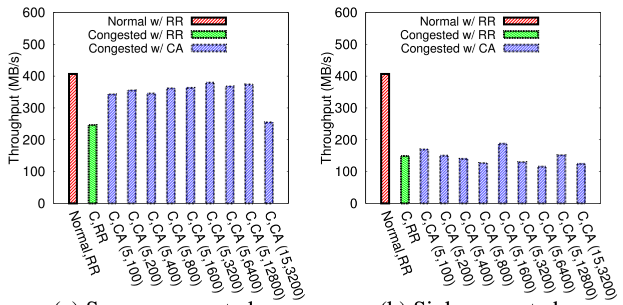
**Impact of RMA buffer size in *LADS*:** All the experiments in the preceding subsections were done by utilizing a large, fixed amount of DRAM (256 MB) for use as RMA buffers at both the source and sink. Given that DTNs are shared resources and multiple users may be using them concurrently, we want to understand what amount of buffering is necessary.

Figure 9 shows the impact of available RMA buffer sizes at the source and sink on *LADS*. We ran each test multiple times and again the variability was very small. As expected, a larger RMA buffer at the source reduces the waiting time for a slot in the RMA buffer by an I/O thread, which improved data transfer rate from the source. Similarly, a greater size of the RMA buffer at sink can hold more data while I/O threads are busy with writing blocks to OSTs, later reducing the time of an I/O thread has to wait until data are ready to be written from the RMA buffer. Interestingly, with the RMA buffer size increasing, *LADS*'s performance does not always improve. Specifically, we have the following observations: (i) a few RMA buffer slots (a few Megabytes) at sink are sufficient to reach the maximum data transfer rate, and (ii) with the increased RMA buffer at source, *LADS* performance improves. It is because at sink, we allow buffered I/Os, thus writes to disks can be fast, whereas at the source, disk read bandwidth is the bottleneck as



(a) Big File Set (b) Small File Set

Figure 9: Impact of RMA buffer size on *LADS*.



(a) Source congested (b) Sink congested

Figure 10: Comparing average run times of transferring 100 x 1 GB files under normal and congested conditions. Source and sink processes are run with eight I/O threads.

DIRECT I/Os are used for data read on the storage at source. It would be beneficial to let multiple I/O threads read data blocks in parallel into the RMA buffer using multiple slots in the RMA buffer. Therefore, it would be more beneficial to add more RMA slots at the source to improve the data read performance than to increase the RMA buffer size at the sink. We also observe that from the big data set test, a smaller RMA buffer size at sink can be the bottleneck, which never happens in the small data set test. We suspect it may be due to the fact that the small files have a `close()` call after each object is transferred which requires a round-trip to the meta-data server, but we did not investigate further.

We also studied scaling performance of *LADS* and bbcp by increasing the number of source-sink instances. We observed that *LADS* outperforms bbcp in aggregate throughput as the number of paired-instances increases. The detail results can be found in our technical report [16].

## 4.3 Congestion-aware I/O Scheduling in Congested Environment

In the previous section, we showed the effectiveness of object scheduling compared to file-based scheduling. In this section, we show the effectiveness of a congestion-aware scheduling algorithm on top of object scheduling in *LADS* for variable I/O load environment on storage systems.

Figure 10 shows the run time comparison results of transferring a total of 100 GB of data in both a normal and storage-congested environment. We executed multiple runs for each test, however there was very little variability in measurement between runs. In the figure, "Normal" indicates when there are no congested disks, "C" means a condition where there are congested disks, and "RR" and "CA" represent *Round Robin* and *Congestion-Aware* scheduling algorithms respectively. In (A, B), A means a threshold to determine if disks are congested, and B denotes a number of times the I/O threads skip one or more disks. To simulate congestion, we used a Linux I/O load generator which uses `libaio` [22]. It generates sequential read requests to four disks with an iteration of five seconds, issuing enough requests to generate 310-350 MB/s of I/O. It runs 10 iterations before it moves on to the next four disks. We had the I/O load generator issue 4 MB requests with a queue depth of four.

For Figure 10(a), we tested various parameter settings, to see the effectiveness of our CA algorithm when the source storage is partially in congestion. Overall, we see that the CA performance can improve by 35% over the RR performance when experiencing congestion. The ranges of a performance improvement can be determined in a function of the threshold, and the number of skips over congested servers. We notice that if the threshold value is set too large or if the number of skips for congested servers to be set either too small or too large, the algorithm likely makes false-positive decisions, negating the performance gain from avoiding congested disks.

For Figure 10(b) shows the results for congestion at the sink PFS. Overall, the performance impact is much significantly higher than when source servers are congested. Surprisingly, the congestion-aware scheduling is almost never improving performance, showing execution times as high as those obtained with the RR algorithm. Irrespective of tuning parameter values, the run times are quite random, mainly because our scheduling algorithm failed to detect congested servers. The congestion-aware algorithm measures I/O service time for each object, but our use of buffered I/Os prevented it from accurately measuring the OSTs' actual level of congestion. We confirmed from our evaluation that most of predictions were false positives, often wrongly assigning I/O threads to busy or overloaded OSTs.

We measured the throughput of bbcp for a congested condition in the storage. The results are shown in Table 1 to compare against the results of *LADS*. We executed multiple runs for each test, however there were very little variability in measurement between runs. The same test-scenarios is used for the *LADS* evaluation presented in Figure 10. It is not surprising that *LADS* is faster than bbcp in both normal and congested conditions. Interestingly, we note that the bbcp run times when the sink is congested are not much different from those under normal conditions, which is most likely due to combination of the OS buffer cache and bbcp's slower communication throughput. It is obvious that buffered I/Os for writes should have been able to hide disk write latency. On the other hand, we observe that bbcp's run time, when the source is experiencing congestion, can increase by 19% over when normal condition. Moreover, bbcp's use of sockets incurs additional copies, user-to-kernel context switches, as well as TCP/IP stack processing. The slower network throughput masks the sink disk congestion. *LADS* clearly benefits from utilizing zero-copy networks when available.

| bbcp | Uncongested Condition | Congested (Side) | |
|---|---|---|---|
| | | Source | Sink |
| Runtime | 21m53s | 26m11s | 21m54s |
| Throughput (MB/s) | 78 | 65 | 78 |

Table 1: Run times and throughput for bbcp under normal and congested environment.

## 4.4 Source-based Buffering using Flash in Congested Environment

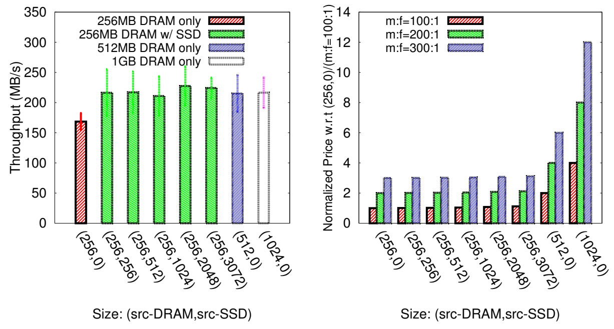
In the previous subsection, we observed that *LADS*' data transfer throughput significantly drops when the sink is overloaded. In this case, the source's RMA buffer becomes full, which stalls the I/O threads from reading additional objects. Therefore, we propose a source-based buffering technique that uses flash-based storage. This source-based SSD buffering utilizes available buffers on flash, which are slower than DRAM yet faster than HDD, to load ahead data blocks to be transferred.

To evaluate it, we slightly modified the overloading workload as we used for Figure 10(b) by inserting ten seconds of idleness between storage congestion periods. During this congestion-free period at sink, source can copy the buffered data from SSD buffer to network RMA buffer. For a fair evaluation, the sink host is set to use only 256 MB RMA buffer, and source and sink run eight I/O threads. The source and sink do not employ the congestion-aware algorithm.

Figure 11(a) shows the results of the effectiveness of the source-based buffering technique using flash. We observe that throughput increases as the available memory for communications at the source increases. However, referring to Figure 11(b), doubling the size of DRAM is very expensive and the same throughput could be achieved using cheaper flash memory.

## 4.5 Production Environment

**DTN to DTN evaluations at ORNL:** To evaluate large-scale performance, we compare the times for transferring a big data set from atlas1 to atlas2 via two DTNs available at ORNL using both *LADS* and bbcp. For this experiment, both bbcp and *LADS* use sockets (in the context of *LADS*, CCI is setup to use its TCP transport) over IPoIB between the source and sink DTNs. The overhead of CCI implementation is quite minimal [3] in which CCI added 150-450 ns to small message latency and no perceptible impact on throughput. On the Lustre Atlas file systems, 1 MB stripe size and a stripe count of four are the default. We ran the experiments twice for every test and Table 2 shows the average throughput (in MB/s). We also want to remind that the Atlas file systems do not use SSDs for buffering.



(a) Throughput (b) Price

Figure 11: Performance analysis of SSD-based object buffering at source. In (a), we showed average throughput with 95% confidence intervals in error bars. In (b), $m:f$ denotes the price ratio between DRAM and Flash.

| Threads (#) | 1 | 2 | 4 | 8 |
|---|---|---|---|---|
| *LADS* | 58.71 | 116.30 | 228.38 | 407.02 |
| bbcp | 59.91 | 58.46 | 57.85 | 59.49 |

Table 2: Throughput comparison (MB/s).

Table 2 presents the data transfer times for *LADS* and bbcp by increasing the number of threads. We observe that throughput when using *LADS* increases with respect to the increased number of I/O threads, whereas adding streams does not help bbcp. With eight threads, *LADS* shows 6.8 times higher data transfer rate than bbcp. However, bbcp shows slightly higher in throughput than *LADS* for a single thread. As we observed earlier, bbcp's single I/O thread issues larger reads that Lustre converts to multiple object reads, while *LADS*' single I/O thread will only read a single object at a time. I/O parallelism for bbcp is limited to four, which is a Lustre default file stripe count. On the other hand, *LADS* allows multiple I/O threads to operate on multiple objects from differing files, resulting in multiple threads to work on multiple OSTs simultaneously. Therefore, *LADS* can fully take advantage of the parallelism available from multiple object storage targets.

## 5 Related Work

Many prior studies have performed on the design and implementation of bulk data movement frameworks [2, 12, 29, 27, 26, 30] and their optimization in wide-area networks [34]. GridFTP [2], provided by Globus toolkit, extends the standard File Transfer Protocol (FTP), and provides high speed, reliable, and secure data transfer. It has a striping feature that enables multi-host to multi-host transfers with each host transferring a subset of files, but does not try to schedule based on the underlying object locations. bbcp [12] is another data transfer utility for moving large files securely, and quickly using multiple streams. It uses a single I/O thread and a file based I/O, and its I/O bandwidth is limited by the stripe width of a file. XDD [29] optimizes the disk I/O performance; enabling file access with direct I/Os and multiple threads for parallelism, and varying file offset ordering to improve I/O access times. These tools are useful for moving large data faster and securely from source host to remote host over the network, but none try to schedule based on the underlying object locations or to detect congested storage targets. Other related work has focused on coupling MPI applications over a terabit network infrastructure [33]. It has investigated a model based on MPI-IO and CCI for transferring large data sets between two MPI applications at different sites. This work does not exploit the underlying file system layouts to improving I/O performance for data transfers either.

Storage contention problems remain a challenge for shared file systems [20, 18, 38, 15, 19]. Reads or writes can be stalled at the file system with overloaded storage targets. The storage target can be busy due to a heavy I/O load by some other applications, or when it is part of a RAID rebuild process. Moreover, I/O server (OSS) can experience bursty I/O requests. Consequently, a longer latency from the storage target can violate the Quality-of-Service (QoS) for file operations [11]. The storage contention can occur even if there is one user application. Multiple threads can implement the program, and one or more threads can share the same storage target, causing contentions. Therefore, the program needs a mechanism for multiple accesses to not compete for the same resource, and needs to be designed in a way to minimize the side-effect created by another user I/O streams. In our prior work [15], we examined the I/O performance of traditional versus layout-aware scheduling. And we addressed a few heuristic algorithms to avoid congested servers. However those algorithms were not fully exploited. On the other hand, in this work, we have fully developed an end-to-end data transfer tool using CCI integrated with layout-awareness algorithms and evaluated them.

Our work differs in several key areas from prior works: (i) We use layout-aware data scheduling to maximize parallelism within the PFS' network paths, servers, and disks. (ii) We focus on the total workload of objects without artificially synchronizing on logical files. (iii) We detect server congestion to minimize our impact on the PFS in order to avoid negatively impacting the performance of the PFS' primary customer, a large HPC system. (iv) We use a modern network abstraction layer, CCI, to take advantage of HPC interconnects to improve throughput.

While our work has focused on I/O optimization for Lustre file systems, one could add support for other parallel file systems such as GPFS [28], and Ceph [36]. *LADS* needs four pieces of information about a given file: object size, stripe width, IDs of servers, and object offsets held by each server. If the parallel file system exposes this information to the user, *LADS* could be implemented for that file system. In Ceph [7] for example, it can return a structure of file system data layout using ioctl with some parameters (e.g., CEPH_IOC_GET_LAYOUT). In Eshel et. al. [8], they describe how pNFS used the layout information of a file in PanFS to perform direct and parallel I/Os.

## 6 Conclusion

Moving large data sets between geographically distributed organizations is a challenging problem which constrains the ability of researchers to share data. Future terabit networks will help improve the network portion of the data transfer, but not the end-to-end transfer, which sources and sinks the data sets in parallel file systems, due to the impedance mismatch between the faster network and much slower storage system. In this study, we identified multiple bottlenecks that exist along the end-to-end data transfer from source and sink host systems in terabit networks, and we proposed *LADS* to demonstrate techniques that can alleviate some end-to-end bottlenecks while at the same time trying not to negatively impact the use of the PFS by other resources, especially large HPC systems. To minimize the effects of transient congestion within a subset of storage servers, *LADS* implemented three I/O optimization techniques: layout-aware scheduling, congestion-aware scheduling, and object caching using SSDs.

## Acknowledgments

We thank the reviewers and our shepherd, Nitin Agrawal, for their constructive comments that have significantly improved the 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] ALCF. Argonne Leadership Computing Facility. `https://www.alcf.anl.gov/`.

[2] ALLCOCK, W., BRESNAHAN, J., KETTIMUTHU, R., LINK, M., DUMITRESCU, C., RAICU, I., AND FOSTER, I. The Globus Striped GridFTP Framework and Server. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2005), SC '05, pp. 54–64.

[3] ATCHLEY, S., DILLOW, D., SHIPMAN, G. M., GEOFFRAY, P., SQUYRES, J. M., BOSILCA, G., AND MINNICH, R. The Common Communication Interface (CCI). In *Proceedings of the Hot Interconnects* (2011), pp. 51–60.

[4] BADAM, A., AND PAI, V. S. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011), NSDI'11, pp. 211–224.

[5] BILL HARROD. US Department of Energy Big Data and Scientific Discovery. `http://www.exascale.org/bdec/sites/www.exascale.org.bdec/files/talk4-Harrod.pdf`.

[6] CCI: Common Communication Interface. `http://cci-forum.com//`.

[7] Ceph. `https://github.com/ceph/`.

[8] ESHEL, M., HASKIN, R., HILDEBRAND, D., NAIK, M., SCHMUCK, F., AND TEWARI, R. Panache: A Parallel File System Cache for Global File Access. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (2010), FAST '10, pp. 155–168.

[9] ESSEN, B. V., HSIEH, H., AMES, S., AND GOKHALE, M. DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, Salt Lake City, UT, USA, November 10-16, 2012* (2012), pp. 731–735.

[10] FUSION-IO. Fusion-io ioDrive Duo. `http://www.fusionio.com/products/iodrive-duo`.

[11] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 7th ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), SIGMETRICS '07, pp. 13–24.

[12] HANUSHEVSKY, A. BBCP. `http://www.slac.stanford.edu/~abh/bbcp/`.

[13] HOLLAND, M., AND GIBSON, G. A. Parity Declustering for Continuous Operation in Redundant Disk Arrays. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (1992), ASPLOS V, pp. 23–35.

[14] HONEYMAN, P., LEVER, C., MOLLOY, S., AND PROVOS, N. The Linux Scalability Project. Tech. rep., 1999.

[15] KIM, Y., ATCHLEY, S., VALLÉE, G. R., AND SHIPMAN, G. M. Layout-Aware I/O Scheduling for Terabits Data Movement. In *Proceedings of the IEEE International Conference on Big Data - Workshop on Distributed Storage Systems and Coding for BigData* (2013), IEEE Big Data '13, pp. 44–51.

[16] KIM, Y., ATCHLEY, S., VALLÉE, G. R., AND SHIPMAN, G. M. LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling. Tech. Rep. ORNL/TM-2014/251, Oak Ridge National Laboratory, Oak Ridge, TN, January 2015.

[17] LIM, S.-H., HUH, J.-S., KIM, Y., SHIPMAN, G. M., AND DAS, C. R. D-factor: A Quantitative Model of Application Slow-down in Multi-resource Shared Systems. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), SIGMETRICS '12, pp. 271–282.

[18] LIU, Q., PODHORSZKI, N., LOGAN, J., AND KLASKY, S. Runtime I/O Re-Routing + Throttling on HPC Storage. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems* (2013), HotStorage '13.

[19] LIU, Y., GUNASEKARAN, R., MA, X., AND VAZHKUDAI, S. S. Automatic Identification of Application I/O Signatures from Noisy Server-Side Traces. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST '14, pp. 213–228.

[20] LOFSTEAD, J., ZHENG, F., LIU, Q., KLASKY, S., OLDFIELD, R., KORDENBROCK, T., SCHWAN, K., AND WOLF, M. 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* (2010), SC '10, pp. 1–12.

[21] NERSC. National Energy Research Scientific Computing Cente. `https://www.nersc.gov/`.

[22] OLCF. I/O Benchmark Suite. `https://www.olcf.ornl.gov/center-projects/file-system-projects/`.

[23] OLCF. Oak Ridge Leadership Computing Facility. https://www.olcf.ornl.gov/.

[24] OPENNVM. OpenNVM. http://opennvm.github.io/.

[25] ORAL, S., SIMMONS, J., HILL, J., LEVERMAN, D., WANG, F., EZELL, M., MILLER, R., FULLER, D., GUNASEKARAN, R., KIM, Y., GUPTA, S., TIWARI, D., VAZHKUDAI, S. S., ROGERS, J. H., DILLOW, D., SHIPMAN, G. M., AND BLAND, A. S. Best Practices and Lessons Learned from Deploying and Operating Large-scale Data-centric Parallel File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (2014), SC '14, pp. 217–228.

[26] REN, Y., LI, T., YU, D., JIN, S., AND ROBERTAZZI, T. Design and Performance Evaluation of NUMA-aware RDMA-based End-to-end Data Transfer Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2013), SC '13, pp. 48:1–48:10.

[27] REN, Y., LI, T., YU, D., JIN, S., ROBERTAZZI, T., TIERNEY, B. L., AND POUYOUL, E. 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* (2012), SC '12, pp. 34:1–34:11.

[28] SCHMUCK, F., AND HASKIN, R. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST '02.

[29] SETTLEMYER, B., DOBSON, J. M., HODSON, S. W., KUEHN, J. A., POOLE, S. W., AND RUWART, T. M. A Technique for Moving Large Data Sets over High-Performance Long Distance Networks. In *Proceedings of the IEEE Symposium on Massive Storage Systems and Technologies* (2011), MSST '11, pp. 1–6.

[30] SUBRAMONI, H., LAI, P., KETTIMUTHU, R., AND PANDA, D. K. High Performance Data Transfer in Grid Environment Using GridFTP over InfiniBand. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing* (2010), CCGRID '10, pp. 557–564.

[31] TORRELLAS, J. Architectures for Extreme-Scale Computing. *Computer 42*, 11 (Nov. 2009), 28–35.

[32] U.S. DEPARTMENT OF ENERGY, OFFICE OF SCIENCE. Energy Science Network (ESnet). http://www.es.net/.

[33] VALLÉE, G., ATCHLEY, S., KIM, Y., AND SHIPMAN, G. M. End-to-End Data Movement Using MPI-IO Over Routed Terabits Infrastructures. In *Proceedings of the 3rd IEEE/ACM International Workshop on Network-aware Data Management* (2013), NDM '13, pp. 9:1–9:8.

[34] VAZHKUDAI, S., SCHOPF, J. M., AND FOSTER, I. F. Predicting the Performance of Wide Area Data Transfers. In *Proceedings of the IEEE 15th International Parallel and Distributed Processing Symposium* (2001), IPDPS '01.

[35] WANG, C., VAZHKUDAI, S. S., MA, X., MENG, F., KIM, Y., AND ENGELMANN, C. NVMalloc: Exposing an Aggregate SSD Store As a Memory Partition in Extreme-Scale Machines. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium* (2012), IPDPS '12, pp. 957–968.

[36] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (2006), OSDI '06, pp. 307–320.

[37] WELCH, B., UNANGST, M., ABBASI, Z., GIBSON, G., MUELLER, B., SMALL, J., ZELENKA, J., AND ZHOU, B. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST '08, pp. 2:1–2:17.

[38] XIE, B., CHASE, J., DILLOW, D., DROKIN, O., KLASKY, S., ORAL, S., AND PODHORSZKI, N. Characterizing Output Bottlenecks in a Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC '12, pp. 8:1–8:11.