Optimizing End-to-End Big Data Transfers over Terabits Network Infrastructure

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

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, for instance, the 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 for terabit networks, called *LADS*. *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 benefit from hardware-level zero-copy, and operating system bypass capabilities 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 input/output bandwidth, and data transfer rates across the high speed networks. We also investigate the performance degradation problems of *LADS* due to I/O contention on the parallel file system (PFS), when multiple *LADS* tools share the PFS. We design and evaluate a meta-scheduler to coordinate multiple I/O streams while sharing the PFS, to minimize the I/O contention on the PFS. With our evaluations, we observe that *LADS* with meta-scheduling can further improve the performance by up to 14% relative to *LADS* without meta-scheduling.

Index Terms—File and Storage systems, Parallel File Sysetms, Networks, I/O Scheduling

1 Introduction

While "Big Data" is now in vogue, many U.S. Department of Energy (DOE) science facilities have produced a vast amount of experimental and simulation data for many years. Several DOE 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 [1]. The Big Data and Scientific Discovery report from the DOE, Office of Science, Office of Advanced Scientific Computing Research (ASCR) [2], 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 from Oak Ridge National Laboratory (ORNL).

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) [3], 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 uses 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.

Y. Kim is currently with the Graduate School of Computer Engineering at Ajou University, Suwon, South Korea e-mail: youkim@ajou.ac.kr. S. Atchley, G. Vallée and S. Lee are with the Oak Ridge National Laboratory, Oak Ridge, TN 37831 USA e-mail: ({atchleyes, valeegr, lees4}@ornl.gov). G. Shipman is with the Los Alamos National Laboratory, Los Alamos, NM 87545 USA e-mail: gshipman@lanl.gov.

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 [4], [5]. Moreover, disk services can degrade while disks in the redundant array of independent disks (RAID) are rebuilding due to failed disks [6]. Also, I/O load imbalance is a serious problem in parallel storage systems [7]. 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) [8], [9], 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 [10], [11], [12], [13], [14] 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 [10]. 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.

Moreover, we further identify the problems of I/O contention on the PFS. Many HPC facilities provide multiple DTNs mounted on the shared PFS. When multiple DTNs are actively using the shared PFS, each data transfer service tool can suffer from the degraded bandwidth due to I/O contention on the shared storage. Thus, we design a meta-scheduler that can coordinate multiple data transfer services' accesses to the PFS, and we comprehensively evaluate the performance improvement by using the meta-scheduler when multiple *LADS* processes are active concurrently.

2 Background

When moving between two PFS at separate sites, the data traverses one or more networks. Over the widearea network (WAN), the data travels via high-bandwidth networks such as DOEs ESnet and Internet2. If a PFS is not directly connected to the WAN, the data additionally transits through one or more local networks.

2.1 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

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.

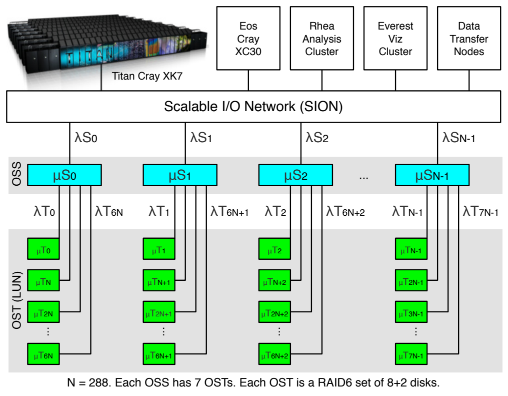


Fig. 1. OLCF center-wide PFS and clients

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 object storage servers (OSSes) in Lustre file systems. For OSS_m, if the arrival rate (λ_{OSSm}) is greater than its service rate (μ_{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 λ_{OSSm} is smaller than μ_{OSSm}, OSTs can become the bottleneck. For example, if λ_{OSTj} is greater than μ_{OSTj}, 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 objects are stored. Figure 2 shows how the OST-to-OSS mapping can physically impact a file's objects placement. The default mapping is to assign OSTs sequentially to OSSes. For a file with a stripe count of 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

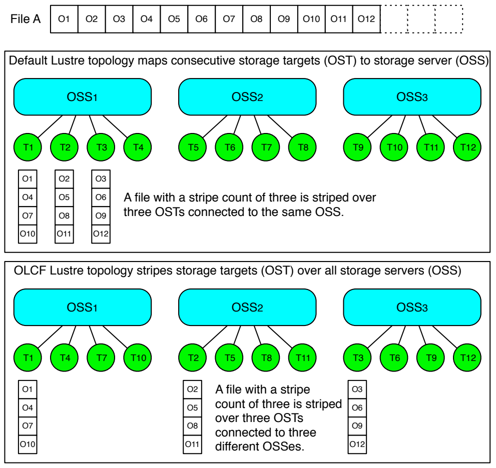

File A
Default Lustre topology maps consecutive storage targets (OST) to storage server (OSS)
A file with a stripe count of three is striped over three OSTs connected to the same OSS.
OLCF Lustre topology stripes storage targets (OST) over all storage servers (OSS)
A file with a stripe count of three is striped over three OSTs connected to three different OSSes.

Fig. 2. File striping in Lustre.

transfer tools [10], [11] 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, respectively causing server or disk contention. 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 with each other on accessing the same OST. Based on a dilation factor model [15], as T_1 and T_2 compete OST_3 , T_1 and T_2 can slow down by 25% and 12.5% respectively. In Figure 3(b), all four threads access different logical regions of the same $File_b$, however, as T_1 and T_2 compete for OST_1, and T_3 and T_4 compete for OST_3. Thus, each thread slows down by 50%. The results of this example indicates that OST contention may increase due to 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*considers 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.

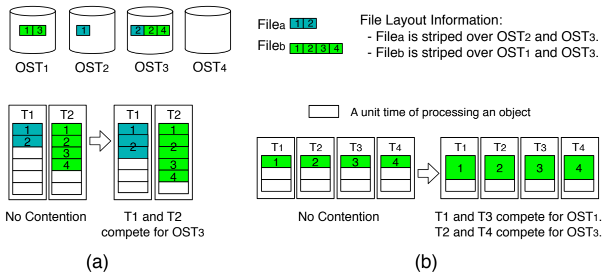

OST1
OST2
OST3
OST4
Filea
Fileb
File Layout Information:
- Filea is striped over OST2 and OST3.
- Fileb is striped over OST1 and OST3.
A unit time of processing an object
No Contention
T1 and T2 compete for OST3
No Contention
T1 and T3 compete for OST1. T2 and T4 compete for OST3.
(a)
(b)

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

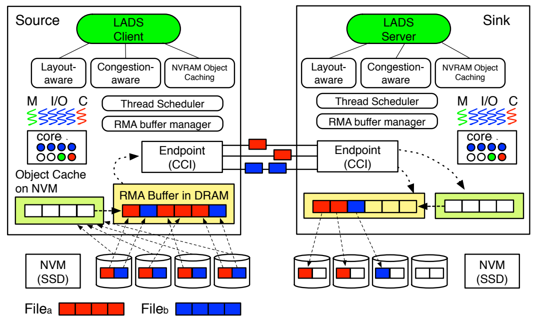


Fig. 4. An architecture overview.

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, combining a threading and an event-driven model.

3.1 LADS Overview

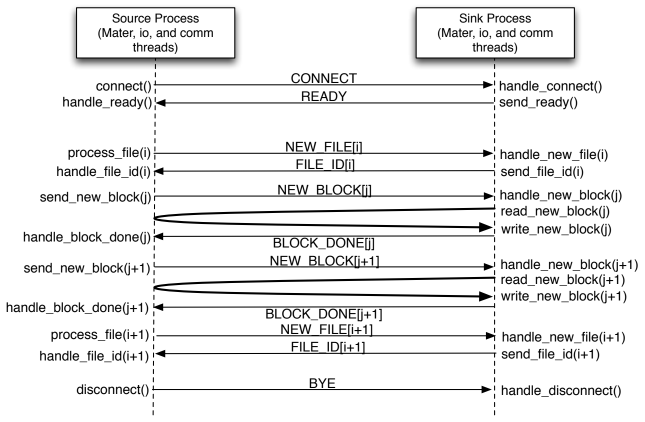
System Architecture: Figure 4 provides an overview of our design and implementation for I/O sourcing and sinking for a PFS. *LADS* is composed of the following threads: The *Master* thread maintains transfer state, while *I/O* threads read and write objects of files from and to the PFS. The *Comm* thread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

Several I/O optimization techniques are implemented in *LADS*. A 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 improve the stalled I/O time when the server is congested. The OST congestion-aware algorithm can avoid the congested servers. Non volatile memory (NVM) can be used as an extended memory region, when the RMA buffer full using the object caching technique. When the RMA buffer is full, I/O threads are blocked and wait until RMA buffer is freed. In order to reduce the blocking time of I/O threads, we propose to implement NVM as an extended memory region using the object caching technique. This technique allows to use the NVM region as an extended buffer region for object caching by I/O threads. 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 notifies 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 only used to synchronize the threads at startup and shutdown as well as to manage the number of files opened and completed. This lock is never accessed in the fast path (i.e., in the Comm or I/O threads). 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 solid-state drive (SSD) as NVM 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 protected by different mutexes. 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.



(a) Communication protocol between source and sink

```
typedef enum msg_type {
      CONNECT = 0,          /* Block size, file list size, RMA handle */
      READY,                /* Have FILE_LIST, start sending NEW_FILE */
      NEW_FILE,             /* Starting new file, need FILE_ID */
      FILE_ID,              /* Here is FILE_ID, start sending NEW_BLOCKS */
      NEW_BLOCK,            /* Block is ready for RMA Read */
      BLOCK_DONE,           /* RMA Read is done, recycle this block */
      BYE,                  /* Ready to disconnect */
      FILE_CLOSE            /* File close*/
} msg_type_t;
```

(b) Message Type

Fig. 5. A diagram for protocol design for source and sink communication and various message types that source and sink communication threads use.

To avoid threads spinning on mutexes as well as "thundering herds" [16] 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 master or an I/O thread has no work available, the waiter will acquire the appropriate lock, enqueue itself on the wait list, and then block on its own condition variable. When another thread has new work available, 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.

In this design, we assume a single master thread, a single communication thread, and multiple I/O threads, where the number of I/O threads is the number of available cores but much less than the number of OSTs in the PFS. The I/O threads only compete with each other to find work from the OSTs queues. Given that the number of all threads is much less than the number of queue/lock pairs, we believe that using traditional mutexes are warranted and the wait queues enable fairness. If in the future, that we determine that mutex contention becomes a larger issue, we could switch to lock-free mechanisms with the trade-off of less fairness.

3.3 Object Transfer Protocol

To better understand how source and sink processes interact including connection establishment, termination, and data transmission, we explain our *LADS* protocol design and implementation using Figure 5(a). The figure specifies several LADS functions and message types, which captures the workflow of data transmission from source to sink hosts. Source and sink processes can be implemented by multiple threads. In our implementation, there is a *master* thread, *N* number of *I/O threads* to fully utilize the parallelism in hosts and parallel file systems for sourcing and sinking multiple data streams, and a *communication thread* at data source and sink hosts. A master thread maintains a tree of file descriptors for source and sink, and schedules the I/O threads. I/O threads can access the OST queues which are as many as OSTs available in the file systems. A communication thread will work on a communication queue, which contains requests for file descriptor exchanges and block requests. The I/O thread can access the queue one at a time, and concurrent accesses by multiple threads to queues are synchronized by OST queue locks. Figure 5(b) describes the message types that communication threads use.

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 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 the access to the RMA buffer. The Master thread initializes its work queue, its wait queue, and the wait queue for I/O threads.

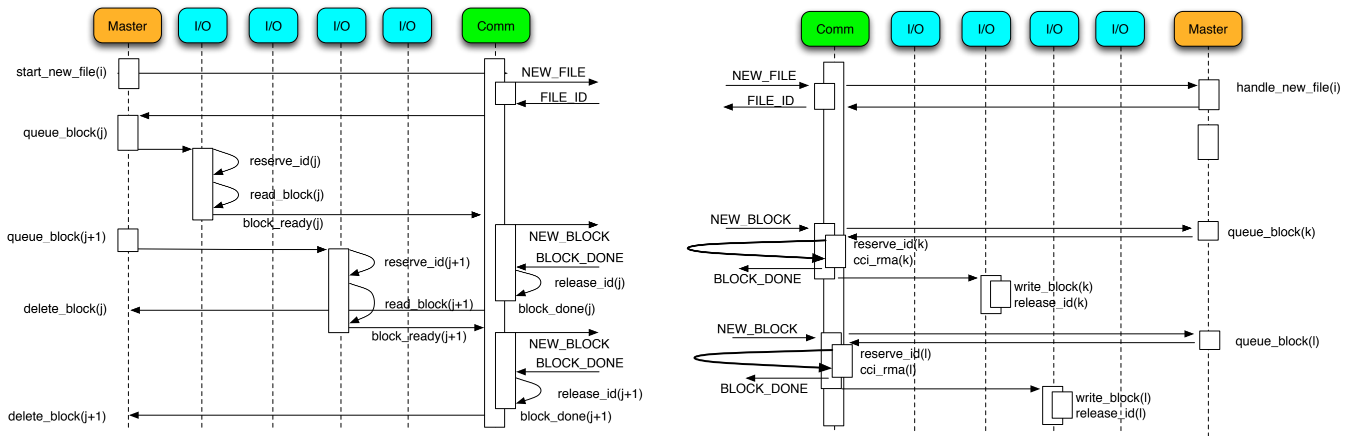
The Comm thread opens a CCI endpoint (send and receive queues, completion queue), allocates and registers with CCI its RMA buffer, and opens a connection to the remote peer. The source Comm thread sends its maximum object size, number of objects in the RMA buffer, and the memory handle for the RMA buffer. 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 the steps as shown in Figure 6 at source and sink.

Step 1. For each file, (i) the source's master opens 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 marshals the `NEW_FILE` request and send it to the sink.

Step 2. At sink, the Comm thread receives the `NEW_FILE` request and enqueues it on the Master's work queue and wakes it up. The Master opens the file, adds the file descriptor to the request, changes the request type to `FILE_ID` and queues the request on the Comm's work queue. The Comm thread dequeues it and sends it to the source.

Step 3. At source, when the Comm thread receives the `FILE_ID` message, it wakes 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`



(a) Source (b) Sink

Fig. 6. Thread communication diagrams at source and sink hosts.

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 contrast, 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 on 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.

Algorithm 1 formally describes the congestion-aware algorithm we developed for *LADS*. Let u be a user (DTN), and let S be the set of object storage targets (OST) stores a set of objects O. Per each OST $s \in S$, a queue q_s maintains information about read or write requests sent to the OST. For simplicity, we denote the location of object o, the OST which contains the object, by $\Phi(o)$. When a user u wants to read a set of files that are composed of a set of objects O_F, as the first step, every queue for each OST is initialized. After initialization, each I/O thread performs the OST-congestion aware algorithm as described in Algorithm 1.

```
begin
    /* Initialization                                              */
    for $o \in O_F$ do
        $s \leftarrow \Phi(o)$ // get the location of an object
        $skip_s = 0$; // initialize skip counter
        $q_s$.enqueue($o$); // add the object to corresponding
            queue
    end
end
begin
    /* OST-congestion aware algorithm                              */
    while true do
        /* A function getNextOST_RR() returns an OST
            with the round-robin policy                            */
        $s \leftarrow$ getNextOST_RR();
        if $skip_s > 0$ then
            $skip_s = skip_s - 1$;
        end
        else
            /* A function averageReadTime($W$) returns
                average read time by measuring readtime
                for $n$ times in time window $W$. Let $T$ be a
                given threshold                                    */
            if averageReadTime($W$)$>T$ then
                $skip_s$ = $M$; // Set $skip_s$ to skip the OST $s$
                    for $M$ times
            end
            else
                $o \leftarrow q_s$.dequeue();
                read($o$);
            end
        end
        if all per-OST queues are empty then
            break;
        end
    end
end
```

Algorithm 1: Congestion-aware I/O Scheduling.

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 [17], [18], [19], [20]. In this work, we specifically use the *NVMalloc* library [18] 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 this feature at this time. Also, should future interconnects support RMA from NVM, we could avoid the copy as well.

4 Experimental Environment

4.1 Experimental Systems

For the evaluation of *LADS*, we use two experimental environments, without and with server congestion, and our production environment. *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 [21]. 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.

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 [22] 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 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). The maximum IB and I/O bandwidths were measured at 3.2 GB/s and 2.3 GB/s respectively. 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. 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.

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 7 plots a file size distribution in terms of the number of files and the aggregate size of files. We plotted a file size distribution in terms of the number of files and the aggregate size of files [23]. We observed 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 whereas the larger files occupy

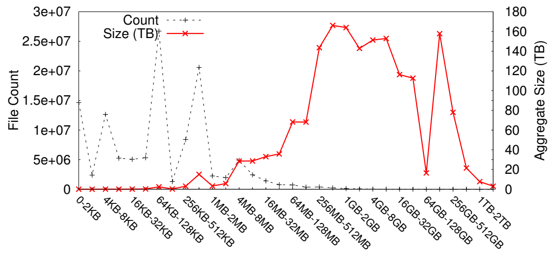


Fig. 7. File size distribution.

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.

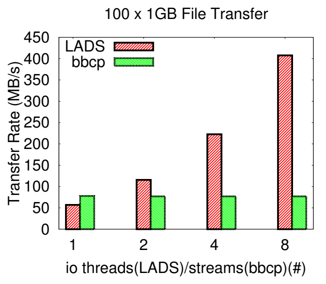
5 Evaluating *LADS*

5.1 Scheduling Objects versus Files in an Uncongested Environment

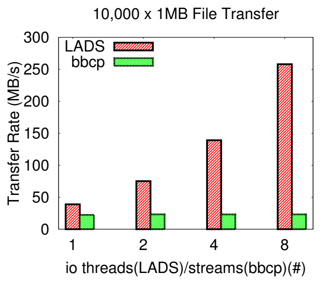
In this section, we show the effectiveness of object scheduling in *LADS* versus file-based scheduling used by bbcp in a controlled, uncongested environment. This section focuses only on the difference between object versus file scheduling; Sections 5.2 and 5.3 will examine two mitigation strategies 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. For big files, 1 GB 100 files are used and for small files, 10,000 1 MB files are used. 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 our prior work [23].

Figures 8 shows 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 only uses a single I/O thread, and for an I/O performance improvement, we have 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. 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.



(a) Big Files



(b) Small Files

Fig. 8. Performance comparisons for *LADS* and bbcp. In bbcp, 10 MB is used for TCP window size.

Performance comparison for object scheduling of *LADS* and file-based scheduling of bbcp: In Figure 8(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 8, 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. *LADS* uses `DIRECT I/O` for the source's read operations to minimize the resource utilization for CPU and memory, while the sink writes using buffered I/O.

We also evaluated resource (CPU and memory) utilization of *LADS* and we witnessed *LADS* moderately uses system resources, and there is only a slight increase in CPU utilization as the number of I/O threads increases. The detail results can be found in our prior work [23].

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 studied the impact of RMA buffer size on the performance of *LADS*. The detail experiment results can be found in our prior work [23].

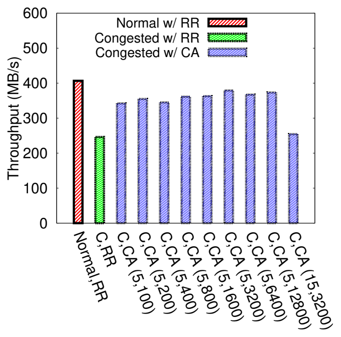
5.2 Congestion-aware I/O Scheduling in Congested Environment

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.

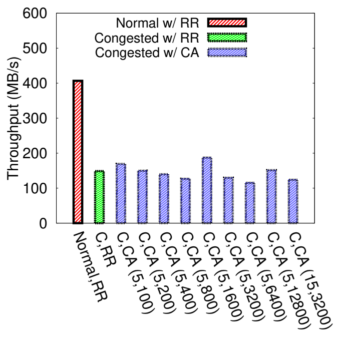
Figure 9 shows the run time comparison results of transferring a total of 100 GB of data in both a normal and storage-congested environment. We executed multiple runs for each test, however there was very little variability in measurement between runs. In the figure, "Normal" indicates when there are 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` [24]. 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 9(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 9(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



(a) Source congested



(b) Sink congested

Fig. 9. 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.

predictions were false positives, often wrongly assigning I/O threads to busy or overloaded OSTs.

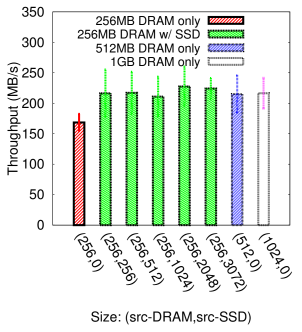
bbcp	Uncongested Condition	Congested (Side)	
		Source	Sink
Runtime	21m53s	26m11s	21m54s
Throughput (MB/s)	78	65	78

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

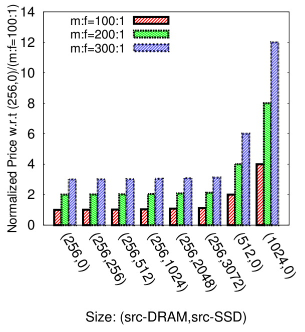
We measured the throughput of bbcp for a congested condition in the storage. The results are shown in Table 1 to compare against the results of *LADS*. The same test-scenarios is used for the *LADS* evaluation presented in Figure 9. We executed multiple runs for each test, however there were very little variability in measurement between runs. 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.

5.3 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,



(a) Throughput



(b) Price

Fig. 10. 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.

to load ahead data blocks to be transferred.

To evaluate it, we slightly modified the overloading workload that we used for Figure 9(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 10(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 10(b), doubling the size of DRAM is very expensive and the same throughput could be achieved using cheaper flash memory.

bbcp's single I/O thread issues larger reads that Lustre converts to multiple object reads, while *LADS*' single I/O thread only reads 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.

We also evaluated large-scale performance of *LADS*, and we compared the times for transferring a big data set from atlas1 to atlas2 via two DTNs available at ORNL using both *LADS* and bbcp. We observed that *LADS* can fully take advantage of the parallelism available from multiple object storage targets, showing significantly improved throughput compared to bbcp. The detail experiment results can be found in our prior work [23].

6 Evaluating *LADS* Tools for Multiple Pairs of Source and Sink Hosts

6.1 I/O Contention from Multiple *LADS* Services

I/O contention does occur when multiple DTNs access the PFS concurrently. In this section, we first illustrate this problem with scaling experiments when multiple *LADS* processes work concurrently. In order to see how much I/O contention degrades the throughput of the concurrently active *LADS* tools, we ran experiments by increasing the number of *LADS* tools in use, where each *LADS* uses 32 I/O threads, and sends *big file workloads* The files at the source are placed in a round robin manner across the OSTs in the PFS.

In Figure 11, we compare the performance of *LADS* with respect to the increased number of source-sink *LADS* pairs. It shows the average per-pair throughput decreases as the number of *LADS* pairs increases. Standard deviations are shown in error bars. Although aggregate throughput does increase, there are diminishing returns. For example, when there is only one *LADS* tool, exclusively accessing the PFS, it can achieve 400-450MB/s of throughput. However, if four *LADS* tools share the PFS and they have to complete the PFS, each *LADS*'s throughput drops below half of an ideal bandwidth without sharing the PFS. We also observe starvation in which one *LADS* tool finishes much faster than the others. For the experiment of 2 pairs of *LADS*, the first *LADS* transferred data at 261 MB/s and while the other transferred at 357 MB/s. By executing multiple *LADS* on multiple DTNs, each *LADS* competes for the PFS causing contention and reduced per-*LADS* throughput.

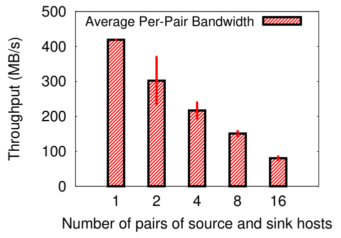


Fig. 11. Average per-pair throughput for *LADS* tool.

6.2 *LADS* Meta-Scheduling for Multiple *LADS* tools

In order to avoid I/O contention on the PFS between *LADS* tools, we design and evaluate a *LADS* meta-scheduler, which defines an access control protocol to the PFS between *LADS* tools.

The *LADS* meta-scheduler follows a client/server model, where the *meta-server* runs on a head node, making scheduling decisions, and a *meta-client* is embedded in each *LADS* tool to consult with the meta-server to request an I/O access privilege. The *meta-server* maintains resource (e.g., OST) privilege vectors for *LADS* tool clients, and gives information about which OSTs in the PFS can be accessed or not.

We first illustrate our proposed algorithm for the Meta-scheduler with the following term definitions. Let U be the set of users (DTNs), and let S be the set of object storage targets (OST) which store objects $o \in O$. We simply denote the location of object o by $\Phi(o)$. The *meta-server* m maintains a set R_{s_i} and another set P_{s_i} for each OST s_i. R_{s_i} contains the information of requested objects, and the P_{s_i} contains the privileged users who are

```
/* Meta-client $c_j$                                          */
begin
    /* Processing transfer requests                           */
    $O_{request} \leftarrow$ newly requested objects by user $u_j$;
    $C_j$.addSet($O_{request}$);
    if $C_j \neq \phi$ then
        $C_j{}' \leftarrow$ scheduler($C_j$);
        // $C_j{}'$ is a selected subset of $C_j$ by the local
           scheduler
        forall the $o \in C_j{}'$ do
            $R_{\Phi(o)}$.add(<$u_j$,$o$>); // $u_j$ is user of $c_j$
        end
    end
    /* Starting new object transfers                          */
    forall the $o \in C_j{}'$ do
        if $u_j \in P_{\Phi(o)}$ then
            startTransfer($o$);
            $C_j$.remove($o$);/* Privilege confirmed, remove
               from the scheduling waiting list               */
        end
    end
    /* Processing completed transfers                         */
    $O_{finished} \leftarrow$ completely transferred objects;
    forall the $o \in O_{finished}$ do
        $R_{\Phi(o)}$.remove(<$u_j$,$o$>);
    end
end
/* Meta-server $m$                                            */
begin
    /* Updating privilege information                         */
    for $i$=1 to $n$ do
        // $n$ is the number of OST
        $P_{s_i} \leftarrow$ updatePrivilege($R_{s_i}$); // Requests from multiple
           users are globally considered
    end
end
```

Algorithm 2: Meta-Scheduling.

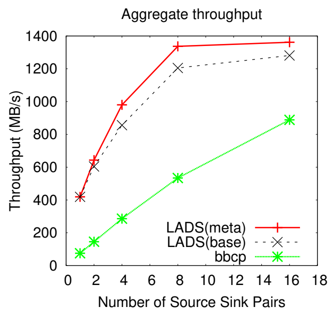
allowed to transfer data from OST s_i. Each *meta-client* c_j of u_j maintains a set C_j, and C_j is composed of objects that are waiting to be scheduled for transfers requested by user u_j.

Algorithm 2 describes the process of each *meta-client* for collaborative data transfers. For a *meta-client* c_j, it has to periodically query the *meta-server* to obtain an OST-access privilege. After an object transfer finishes, it notifies the *meta-server* of object's transfer completion. The *meta-server* then updates the list of OST privileges. *updatePrivilege()* takes R_{s_i}=$\{<u_k,o_k>|k$=1 $\ldots$ $m\}$ as its arguments and returns a set of users $U_{privileged} \subset \{u_1, \ldots, u_k\}$, where the users are allowed to transfer data from the OST s_i. If R_{s_i}=ϕ, it returns ϕ as its output. Note that in the function *updatePrivilege()*, each R_{s_i} is implemented in a queue, and it returns an oldest user from the queue and if the queue is empty, it returns NULL.

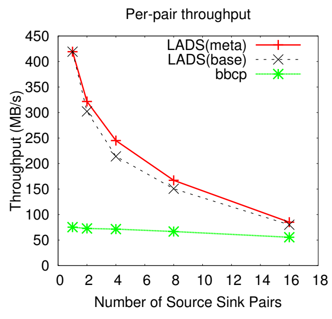
6.3 *LADS* Meta-Scheduler Performance

We performed scalability experiments for *LADS* tools by comparing our proposed meta-scheduling algorithm (*LADS-meta*), *LADS* tools without meta-scheduling (*LADS-base*), and bbcp. For comparison, we measured average aggregate throughput and average per-pair throughput (source-to-sink in a pair) by increasing the number of source-sink instances. For these experiments, we have used *big files* and *small files* workloads.

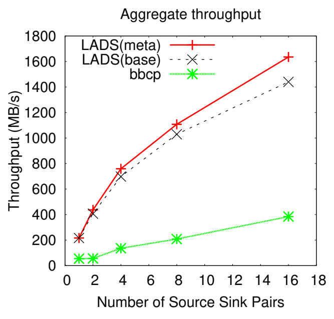
Figures 12(a) & (b) show the results of the *big files* workload when increasing the number of paired-instances to sixteen. In Figure 12(a), we observe that LADS-base significantly outperforms bbcp in aggre-



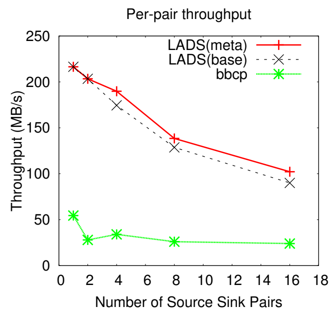
(a)



(b)



(c)



(d)

Fig. 12. Scaling experiments by increasing the number of DTN source and sink pairs. (a)&(b) and (c)&(d) are the results for big files workload and small files workload respectively.

gate throughput as the number of paired-instances increases. At each number of paired-instances, *LADS* outperforms bbcp by a significant amount. LADS-meta shows the improved throughput over LADS-base. At two paired instances, LADS-meta showed an 12.5% improved throughput than LADS-base. The improvement becomes 10.95% at eight paired instances. However, the rate of improvement shows diminishing returns at 16 paired instances. This is because the PFS is overwhelmed by too may I/O simultaneous accesses, so our meta scheduling algorithm cannot find room for scheduling. Figure 12(b) shows that the average per-pair throughput for *LADS* (both LADS-meta and LADS-base) experiences diminishing returns. bbcp, on the other hand, scales nearly linearly up to eight paired-instances, albeit at a lower level than *LADS*, and then it too shows diminishing returns.

We have similar findings for the *small files* workload in which LADS-base significantly outperforms bbcp in terms of both aggregate and per-pair throughput. Interestingly, bbcp's aggregate throughput is not increasing as much as we observed in the big files workload results. We believe this is partly due to the lack of I/O parallelism within bbcp because each file is contained by a single OST. bbcp's per-pair throughput shows a sharp drop from one pair to four pairs possibly due to accessing the same OSTs. LADS-meta further improves the performance of LADS-base by up to 13.5%. At present, current the meta-scheduling algorithm does not guarantee quality of service (QoS) for each *LADS* tool, however, the algorithm described in Algorithm 2 can be easily extended to offer QoS guarantees.

6.4 Integration of *LADS* Meta-Scheduler with HPC Storage Health Monitoring

The *LADS* Meta-scheduler can be integrated with the HPC Storage Health Monitoring Service [25] to enable better scheduling of the shared storage. At present, our Meta-scheduler is only designed to control the PFS access by multiple *LADS* tools to avoid I/O interference to the PFS between them. Thus, it is blind to I/O traffic and usage on the PFS by simulation platforms (e.g., Titan). The health monitoring infrastructure [26], [25] monitors

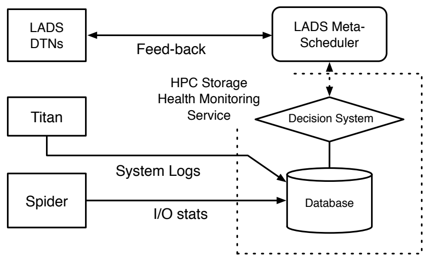


Fig. 13. Integration of *LADS* Meta-Scheduler with HPC Storage Health Monitoring Service.

the file system usage using a custom, in-house built-in tool [26] by querying the RAID controllers for the read/write bandwidth and IOPS data, along with the request size distribution at certain time intervals. The data collected is stored in a Mysql database for offline analysis, and used to predict future I/O access patterns on the PFS. Figure 13 outlines our proposed integration of the *LADS* Meta-Scheduler with the HPC Health Monitoring tool to give the Meta-Scheduler insight into the overall PFS activity levels. To determine the I/O utilization of a particular server or disk, the *LADS* Meta-Scheduler requires information about I/O utilization (referring to bandwidth and IOPS usage information) from the HPC Storage Health Monitoring service. In response to this request, the health monitoring infrastructure can return historical information of I/O usage on the PFS and expected future I/O usage based on the regular I/O patterns of the competing systems' workloads. Therefore, an iterative procedure is proposed to update the resource vectors in the *LADS* Meta-Scheduler with these I/O usage information for each server or disk in the PFS. When the Meta-scheduler is requested from *LADS* tools, it returns with the updated vector information.

7 Related Work

Many prior studies were performed on the design and implementation of bulk data movement frameworks [11], [10], [12], [13], [27], [14]. GridFTP, 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 [10] 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 [12] 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 [9]. 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.

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.

8 Conclusion

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. We designed and evaluated an intra-DTN *LADS* scheduling algorithm to control uncoordinated I/O traffic on the shared PFS, minimizing the impact of the PFS I/O contention on *LADS*. To demonstrate the efficiency of *LADS*, we evaluated it on a controlled test-bed with a 32 disk Lustre file system as well as on production DTN nodes, which mount the Atlas file systems at ORNL. We studied throughput and resource utilization comparisons for *LADS* and bbcp, and demonstrated that *LADS* can better utilize the parallelism available in the PFS. We also evaluated the efficiency of our layout-aware I/O algorithm as well as adding congestion-awareness to it. We showed that the congestion-aware I/O algorithm can provide about 30-40% higher throughput than when the layout-aware algorithm is used alone. We also proposed a source-based SSD buffering technique in order to further improve the performance of *LADS* when sink servers are congested. *LADS* is implemented using CCI, which is portable and can be used in just about any network environment without any modification to the *LADS* prototype. CCI also allows *LADS* to take advantage of networks which provide advanced capabilities, such as zero-copy transfers. Lastly, we comprehensively evaluated our proposed meta-scheduling algorithms to control I/O loads on the PFS between competing *LADS* services. Our evaluation showed the meta-scheduling algorithm further improve per-pair throughput by up to 11% compared to a case without meta-scheduling.

We can extend our research to develop a holistic data transfer framework, which determines timing of data transfers in the system environment where I/O loads on the PFS are dynamically changing. The *LADS* meta-scheduler integrated with the HPC Health Monitoring tool will give the meta-scheduler insight into the overall PFS activity levels, such that it can optimally schedule I/O accesses on the PFS. Moreover, we can extend *LADS* to be fault-tolerant. In the existing file based transfer, on a DTN failure, a current file transfer has to be retransmitted, however, in *LADS*, objects of the incomplete files, which have not been transferred, will only need to be transferred by simply maintaining persistent logs at sink. We plan on developing a fault-tolerant *LADS* in the future work.

Acknowledgments

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No. R0190-15-2012, High Performance Big Data Analytics Platform Performance Acceleration Technologies Development). This research is also 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] Josep Torrellas. Architectures for Extreme-Scale Computing. *Computer*, 42(11):28–35, November 2009.
[2] 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.
[3] U.S. Department of Energy, Office of Science. Energy Science Network (ESnet). http://www.es.net/.
[4] Ajay Gulati, Arif Merchant, and Peter J. Varman. 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*, SIGMETRICS '07, pages 13–24, 2007.
[5] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. Characterizing Output Bottlenecks in a Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 8:1–8:11, 2012.
[6] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 1–17, 2008.
[7] 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, 2010.

[8] Scott Atchley, David Dillow, Galen M. Shipman, Patrick Geoffray, Jeffrey M. Squyres, George Bosilca, and Ronald Minnich. The Common Communication Interface (CCI). In *Proceedings of the Hot Interconnects*, pages 51–60, 2011.
[9] Geoffroy Vallée, Scott Atchley, Youngjae Kim, and Galen M. Shipman. 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*, NDM '13, pages 9:1–9:8, 2013.
[10] Andrew Hanushevsky. BBCP. http://www.slac.stanford.edu/~abh/bbcp/.
[11] 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 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '05, pages 54–64, 2005.
[12] 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 IEEE Symposium on Massive Storage Systems and Technologies*, MSST '11, pages 1–6, 2011.
[13] 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, 2012.
[14] Hari Subramoni, Ping Lai, Raj Kettimuthu, and Dhabaleswar K. Panda. 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*, CCGRID '10, pages 557–564, 2010.
[15] Seung-Hwan Lim, Jae-Seok Huh, Youngjae Kim, Galen M. Shipman, and Chita R. Das. 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*, SIGMETRICS '12, pages 271–282, 2012.
[16] Peter Honeyman, Chuck Lever, Stephen Molloy, and Niels Provos. The Linux Scalability Project. Technical report, 1999.
[17] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, pages 211–224, 2011.
[18] Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma, Fei Meng, Youngjae Kim, and Christian Engelmann. 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*, IPDPS '12, pages 957–968, 2012.
[19] Brian Van Essen, Henry Hsieh, Sasha Ames, and Maya Gokhale. 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*, pages 731–735, 2012.
[20] OpenNVM. OpenNVM. http://opennvm.github.io/.
[21] CCI: Common Communication Interface. http://cci-forum.com/.
[22] Fusion-io. Fusion-io ioDrive Duo. http://www.fusionio.com/products/iodrive-duo.
[23] Youngjae Kim, Scott Atchley, Geoffroy Schmuck, Vallée, and M. Galen Shipman. LADS: Optimizing Data Transfers using Layout-Aware Data Scheduling. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '15, Berkeley, CA, USA, 2015. USENIX Association.
[24] OLCF. I/O Benchmark Suite. https://www.olcf.ornl.gov/center-projects/file-system-projects/.
[25] Raghul Gunasekaran and Youngjae Kim. Feedback Computing in Leadership Compute Systems. In *9th International Workshop on Feedback Computing (Feedback Computing 14)*, Philadelphia, PA, June 2014. USENIX Association.
[26] Youngjae Kim, Raghul Gunasekaran, Galen M. Shipman, and David A. Dillow. Workload Characterization of a Leadership Class Storage. In *PDSW*, 2010.
[27] Yufei Ren, Tan Li, Dantong Yu, Shudong Jin, and Thomas Robertazzi. 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*, SC '13, pages 48:1–48:10, 2013.

Youngjae Kim is currently an assistant professor in the Department of Information and Computer Engineering at Ajou University. He received the B.S. degree in computer science from Sogang University, Korea in 2001, the M.S. degree from KAIST in 2003 and the Ph.D. degree in computer science and engineering from Pennsylvania State University in 2009. He had worked at the Oak Ridge National Laboratory as an I/O Systems Computational Scientist in 2009-2015. His research interests include operating systems, parallel I/O and file systems, storage systems, emerging storage technologies, and performance evaluation for high-performance computer systems.

Scott Atchley received a BS degree in Business Administration and a MS degree in Computer Science from the University of Tennessee in 1987 and 2002, respectively. He joined Oak Ridge National Laboratory as a HPC Systems Engineer in 2011. He is the team lead for System Architecture, Resilience, and Networking in the Technology Integration group within ORNL's National Center for Computational Science. His research interests include high-performance interconnects and their interfaces, system architectures, and multi-level memories. Prior to joining ORNL, he was a member of the technical staff at Myricom and a research leader at the University of Tennessee.

Geoffroy Vallée received the MS degree in computer science from Universite de Versailles Saint-Quentin-en-Yvelines, France in 2000; the PhD degree in computer science from Universite Rennes 1, France in 2004 during which he collaborated with both INRIA and Electricite de France (EDF). He joined Oak Ridge National Laboratory in 2004 as a postdoctoral researcher and became a research scientist in 2007. His research focuses on system for high-performance computing, including operating systems, networking substrates, run-time systems, resilience and fault tolerance.

Sangkeun Lee received the BS degree in computer science from Korea Advanced Institute of Science and Technology in 2005 and the PhD degree in computer science and engineering from Seoul National University in 2012. He joined Oak Ridge National Laboratory as a postdoctoral research associate in 2013. His research interests include large-scale graph mining and analytics, big data systems and architectures, and information retrieval and recommender systems.

Galen M. Shipman joined Los Alamos National Laboratory as a Computer Scientist in 2015. Before joining LANL, he was the Director for Compute and Data Environment for Science at Oak Ridge National Laboratory. He was responsible for defining and maintaining an overarching strategy for data storage, data management, and data analysis spanning from research and development to integration, deployment and operations for high-performance and data-intensive computing initiatives at ORNL. Prior to joining ORNL, he was a technical staff member in the Advanced Computing Laboratory at Los Alamos National Laboratory. Mr. Shipman received his B.B.A. in finance in 1998 and a M.S. degree in computer science in 2005 from the University of New Mexico. His research interests include High Performance and Data Intensive Computing.