

FusionFS: Towards Supporting Data-Intensive Scientific Applications on Extreme-Scale High-Performance Computing Systems

Dongfang Zhao^{*◇}, Zhao Zhang[†], Xiaobing Zhou[‡], Tonglin Li^{*},
Dries Kimpe[◇], Phil Carns[◇], Robert Ross[◇], and Ioan Raicu^{*◇}

^{*}Illinois Institute of Technology [†]University of California, Berkeley [‡]Hortonworks Inc.
[◇]Argonne National Laboratory

dzhao8@iit.edu, zhaozhang@eecs.berkeley.edu, xzhou@hortonworks.com, tli33@hawk.iit.edu,
{dkimpe, carns, ross}@mcs.anl.gov, iraicu@cs.iit.edu

Abstract—State-of-the-art yet decades old architecture of high performance computing (HPC) systems has its compute and storage resources separated. It has shown limits for today’s data-intensive scientific applications, because every I/O needs to be transferred via the network between the compute and storage cliques. This paper proposes a distributed storage layer local to the compute nodes, which is responsible for most of the I/O operations and saves extreme amount of data movement between compute and storage resources. We have designed and implemented a system prototype of such architecture –the FusionFS distributed file system– to support metadata-intensive and write-intensive operations, both of which are critical to the I/O performance of scientific applications. FusionFS has been deployed and evaluated on up to 16K compute nodes in an IBM Blue Gene/P supercomputer, showing more than an order of magnitude performance improvement over other popular file systems such as GPFS, PVFS, and HDFS.

I. INTRODUCTION

The conventional architecture of high-performance computing (HPC) systems separates the compute and storage resources into two cliques (i.e. compute nodes and storage nodes), both of which are interconnected by a shared network infrastructure. This architecture is mainly a result from the nature of many legacy large-scale scientific applications that are compute intensive, where it is often assumed that the storage I/O capabilities are lightly utilized for the initial data input, some periodic checkpoints, and the final output. However, in the era of Big Data, scientific applications are becoming more and more data-intensive, requiring a greater degree of support from the storage subsystem [1]. Our previous simulation work [2] demonstrates that the current HPC storage architecture would not scale to the emerging exascale computing systems (10^{18} ops/s).

While recent studies [3, 4] addressed the I/O bottleneck in the conventional architecture of HPC systems, this paper is orthogonal to them by proposing a new storage architecture to co-locate the storage and compute resources. The ideas in this paper are built upon prior work [5] presented at the 2012 Supercomputing conference as a poster. In particular,

we envision a distributed storage system on compute nodes for applications to manipulate their intermediate results and checkpoints, rather than transferring data over the network. While co-location of storage and computation has been widely leveraged in data centers (e.g. Hadoop clusters), such architecture never exists in HPC systems even though it has attracted much research interest recently, e.g. the DEEP-ER [6] project funded by the European Union. This work demonstrates how to architect and engineer such a system, and reports how much, quantitatively, it could improve the I/O performance of real-world scientific applications.

The proposed architecture of co-locating compute and storage could raise concerns about jitters on compute nodes, since applications’ computation and I/O share resources like CPU and network. We argue that the I/O-related cost can be offloaded onto dedicated infrastructures that are decoupled from the application’s acquired resources, as justified in [7]. In fact, this resource-isolation strategy has been applied in production systems: the IBM Blue Gene/Q supercomputer (Mira [8]) assigns one core of the chip (17 cores in total) for the local operating system and the other 16 cores for applications.

Distributed storage has been extensively studied in data centers (e.g. the popular distributed file system HDFS [9]); yet there exists little literature for building a distributed storage system particularly for HPC systems whose design principles are much different from data centers. HPC nodes are highly customized and tightly coupled with high throughput and low latency network (e.g. InfiniBand), while data centers typically have commodity servers and inexpensive networks (e.g. Ethernet). So storage systems designed for data centers are not optimized for the HPC machines, as we will discuss in more detail where HDFS shows poor performance on a typical HPC machine (Figure 11). In particular, we observe that the following challenges are unique to a distributed file system on HPC compute nodes, related to both metadata-intensive and write-intensive workloads.

First, the storage system on HPC compute nodes needs to support intensive metadata operations. Many scientific applications create a large number of small- to medium-sized files, as Welch and Noer [10] reported that 25% – 90% of all the 600 million files from 65 Panasas [11] installations are 64KB or smaller. So the I/O performance is highly throttled by the metadata rate, besides the data itself. Data centers, however, are not optimized for this type of workload. If we recall that HDFS [9] splits a large file into a series of default 64MB chunks (128MB recommended in most cases) for parallel processing, a small- or medium-sized file can benefit little from this data parallelism. Moreover, the centralized metadata server in HDFS is apparently not designed to handle intensive metadata operations.

Second, file writes should be optimized for a distributed file system on HPC compute nodes. The fault tolerance of most today’s large-scale HPC systems is achieved through some form of checkpointing. In essence, the system periodically flushes memory to external persistent storage, and occasionally loads the data back to memory to roll back to the most recent correct checkpoint up on a failure. So file writes typically outnumber file reads in terms of both frequency and size in HPC systems, and improving the write performance will significantly reduce the overall I/O cost. The fault tolerance of data centers, however, is not achieved through checkpointing its memory states, but the re-computation of affected data chunks that are replicated on multiple nodes.

We have designed and implemented the FusionFS distributed file system to overcome the aforementioned challenges. FusionFS disperses its metadata to all the available compute nodes to achieve the maximal concurrency of metadata operations. Every client of FusionFS optimizes write operations with local writes (whenever possible), which reduces network traffic and makes the aggregate I/O throughput highly scalable. We expect FusionFS to coexist with the remote parallel file system (e.g. GPFS [12]) rather than to replace the latter, because the compute nodes of current HPC systems are tightly coupled and are not viable to provide on-board storage as large as the remote parallel file systems.

FusionFS has been deployed on up to 16K compute nodes of an IBM Blue Gene/P supercomputer (Intrepid [13]), and heavily accessed by a variety of benchmarks and applications. We observed more than an order of magnitude improvement to the I/O performance when comparing FusionFS to other popular file systems such as GPFS [12], PVFS [14], and HDFS [9], surpassing 2.5TB/s aggregate I/O throughput on 16K nodes. In addition, FusionFS has been serving as the infrastructure or test bed of a few related projects such as virtual-chunk-based file compression [15, 16].

In summary, this paper makes the following contributions:

- *Propose an unprecedented storage architecture for extreme-scale HPC systems to address the I/O bottleneck of modern data-intensive scientific applications*
- *Design and implement the FusionFS distributed file system to support metadata-intensive and write-intensive file operations*

- *Evaluate FusionFS with benchmarks and applications at extreme scales, and demonstrate its superiority over state-of-the-art solutions*

II. DESIGN OVERVIEW

As shown in Figure 1, FusionFS is a user-level file system that runs on the compute resource infrastructure, and enables every compute node to actively participate in both the metadata and data movement. The client (or application) is able to access the global namespace of the file system with a distributed metadata service. Metadata and data are completely decoupled: the metadata on a particular compute node does not necessarily describe the data residing on the same compute node. The decoupling of metadata and data allows different strategies to be applied to metadata and data management, respectively.

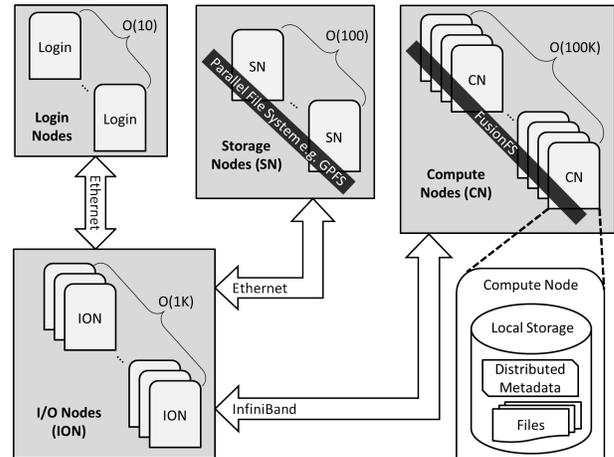


Figure 1. FusionFS deployment in a typical HPC system

FusionFS supports both the POSIX interface and a user library. The POSIX interface is implemented with the FUSE framework [17], so that legacy applications can run directly on FusionFS without modifications. Just like other user-level file systems (e.g. PVFS [14]), FusionFS can be deployed as a mount point in a UNIX-like system. The mount point is a virtual root directory to the clients when using FusionFS.

Users need to specify three arguments when deploying FusionFS as a POSIX-compliant mount point on a compute node: the scratch directory where to store the metadata and data, the mount point of the remote parallel file system (e.g. Lustre [18], GPFS [12], PVFS [14]), and the mount point of FusionFS where applications manipulate files. The remote parallel file system needs to be integral to the global namespace because it is necessary to accommodate large files that cannot fit in FusionFS.

FUSE has been criticized for its performance overhead. In native UNIX-like file systems (e.g. Ext4) there are only two context switches between the user space and the kernel. In contrast, for a FUSE-based file system, context needs to be switched four times: two switches between the caller and VFS; and another two between the FUSE user library (*libfuse*) and the FUSE kernel module (*devfuse*). A detailed comparison

between FUSE-enabled and native file systems was reported in [19], showing that a Java implementation of a FUSE-based file system introduces about 60% overhead compared to the native file system. However, in the context of C/C++ implementation with multithreading on memory-level storage, which is a typical setup in HPC systems, the overhead is much lower. In prior work [20], we reported that FUSE could deliver as high as 578MB/s throughput, 85% of the raw bandwidth.

To avoid the performance overhead from FUSE, FusionFS also provides a user library for applications to directly interact with their files. These APIs look similar to POSIX, for example `ffs_open()`, `ffs_close()`, `ffs_read()`, and `ffs_write()`. The downside of this approach is the lack of POSIX support, indicating that the application might not be portable to other file systems, and often needs some modifications and recompilation.

III. METADATA MANAGEMENT

A. Namespace

Clients have a coherent view of all the files in FusionFS no matter if the file is stored in the local node or a remote node. This global namespace is maintained by a distributed hash table (DHT [21]), which disperses partial metadata on each compute node, and has served as the infrastructure for a few other systems such as data provenance [22, 23] and key-value stores [24]. As shown in Figure 2, in this example Node 1 and Node 2 only physically store two subgraphs (the top left and top right portion of the figure) of the entire metadata graph. The client could interact with the DHT to inquire any file on any node, as shown in the bottom portion of the figure. Because the global namespace is just a logical view for clients, and it does not physically exist in any data structure, the global namespace does not need to be aggregated or flushed when changes occur to the subgraph on local compute nodes. The changes to the local metadata storage will be exposed to the global namespace when the client queries the DHT.

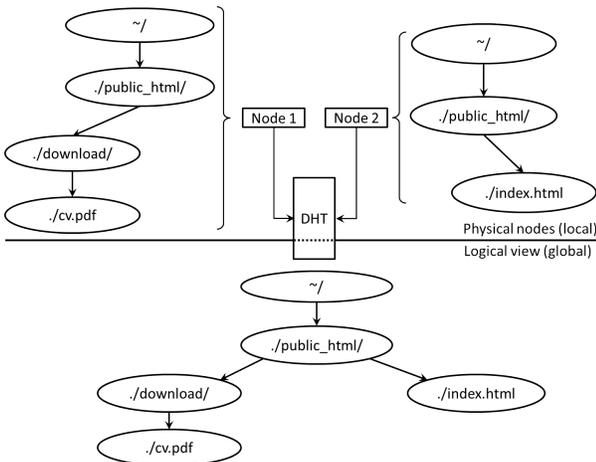


Figure 2. Metadata in the local nodes and the global namespace

B. Data Structures

FusionFS has different data structures for managing regular files and directories. For a regular file, the field `addr` stores the node where this file resides. For a directory, there is a field `filelist` to record all the entries under this directory. This `filelist` field is particularly useful for providing an in-memory speed for directory read, e.g. “`ls /mnt/fusionfs`”. Nevertheless, both regular files and directories share some common fields, such as timestamps and permissions, which are commonly found in traditional i-nodes.

To make matters more concrete, Figure 3 shows the distributed hash table according to the example metadata shown in Figure 2. Here, the DHT is only a logical view of the aggregation of multiple partial metadata on local nodes (in this case, Node 1 and Node 2). Five entries (three directories, two regular files) are stored in the DHT, with their file names as keys. The value is a list of properties delimited by semicolons. For example, the first and second portions of the values are permission flag and file size, respectively. The third portion for a directory value is a list of its entries delimited by commas, while for regular files it is just the physical location of the file, e.g. the IP address of the node on which the file is stored. Upon a client request, this value structure is serialized by Google Protocol Buffers [25] before sending over the network to the metadata server, which is just another compute node. Similarly, when the metadata blob is received by a node, we deserialize the blob back into the C structure with Google Protocol Buffers.

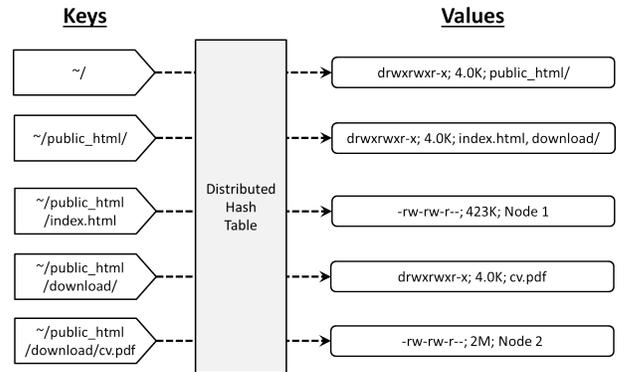


Figure 3. The global namespace abstracted by key-value pairs in a DHT

The metadata and data on a local node are completely decoupled: a regular file’s location is independent of its metadata location. This flexibility allows us to apply different strategies to metadata and data management, respectively. Moreover, the separation between metadata and data has the potential to plug in alternative components to metadata or data management, making the system more modular.

From Figure 2, we know the `index.html` metadata is stored on Node 2, and the `cv.pdf` metadata is on Node 1. However, it is perfectly fine for `index.html` to reside on Node 1, and for `cv.pdf` to reside on Node 2, as shown in Figure 3. Besides the conventional metadata information for regular files, there is a

special flag in the value indicating if this file is being written. Specifically, any client who requests to write a file needs to set this flag before opening the file, and will not reset it until the file is closed. The atomic compare-swap operation supported by DHT [21] guarantees the file consistency for concurrent writes.

Another challenge on the metadata implementation is on the large-directory performance issues. In particular, when a large number of clients write many small files on the same directory concurrently, the value of this directory in the key-value pair gets incredibly long and responds extremely slowly. The reason is that a client needs to update the entire old long string with the new one, even though the majority of the old string is unchanged. To fix that, we implement an atomic append operation that asynchronously appends the incremental change to the value. This approach is similar to Google File System [26], where files are immutable and can only be appended. This gives us excellent concurrent metadata modification in large directories, at the expense of potentially slower directory metadata read operations.

C. Network Protocols

We encapsulate several network protocols in an abstraction layer. Users can specify which protocol to be applied in their deployments. Currently, we support three protocols: TCP, UDP, and MPI. Since we expect a high network concurrency on metadata servers, `epoll` [27] is used instead of multithreading. The side effect of `epoll` is that the received message packets are not kept in the same order as on the sender. To address this, a header `[message_id, packet_id]` is added to the message at the sender, and the message is restored by sorting the `packet_id` for each message at the recipient. This is efficiently done by a sorted map with `message_id` as the key, mapping to a sorted set of the message's packets.

D. Persistence

The whole point of the proposed distributed metadata architecture is to improve performance. Thus, any metadata manipulation from clients should occur in memory, plus some network transfer if needed. On the other hand, persistence is required for metadata just in case of any memory errors or system restarts.

The persistence of metadata is achieved by periodically flushing the in-memory metadata onto the local persistent storage. In some sense, it is similar to the incremental checkpointing mechanism. This asynchronous flushing helps to sustain the high performance of the in-memory metadata operations.

E. Consistency

Since each primary metadata copy has replicas, the next questions is how make them consistent. Traditionally, there are two semantics to keep replicas consistent: (1) strong consistency – blocking until replicas are finished with updating; (2) weak consistency – return immediately when the primary copy is updated. The tradeoff between performance and consistency

is tricky, most likely depending on the workload characteristics.

As for a system design without any *a priori* information on the particular workload, we compromise with both sides: assuming the replicas are ordered by some criteria (e.g. last modification time), the first replica is strong consistent to the primary copy, and the other replicas are updated asynchronously. By doing this, the metadata are strong consistent (in the average case) while the overhead is kept relatively low.

IV. FILE MANIPULATION

A. Network Transfer

For file transfer, neither UDP nor TCP is ideal for FusionFS on HPC compute nodes. UDP is a highly efficient protocol, but lacks reliability support. TCP, on the other hand, supports reliable transfer of packets, but adds significant overhead.

We have developed our own data transfer service Fusion Data Transfer (FDT) on top of UDP-based Data Transfer (UDT) [28]. UDT is a reliable UDP-based application level data transport protocol for distributed data-intensive applications. UDT adds its own reliability and congestion control on top of UDP that offers a higher speed than TCP.

B. File Open

Figure 4 shows the protocol when opening a file in FusionFS. Due to limited space, we assume the requested file is also on Node-j. Note that it is not necessarily Node-j who stores both the requested file and its metadata, as we explained in Section III-B that the metadata and data are decoupled on compute nodes.

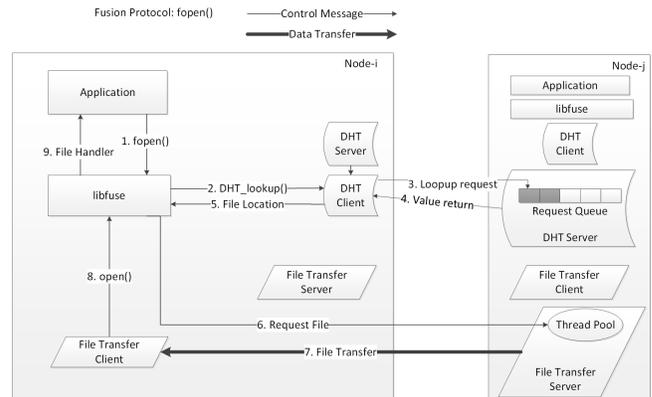


Figure 4. The protocol of file open in FusionFS

In step 1, the application on Node-i issues a POSIX `fopen()` call that is caught by the implementation in the FUSE user-level interface (i.e. `libfuse`) for file open. Steps 2 – 5 retrieve the file location from the metadata service that is implemented by a distributed hash table [21]. The location information might be stored in another machine Node-j, so this procedure could involve a round trip of messages between Node-i and Node-j. Then Node-i needs to ping Node-j to fetch the file in steps 6 – 7. Step 8 triggers the system call to open the

transferred file and finally step 9 returns the file handle to the application.

C. File Write

Before writing to a file, the process checks if the file is being accessed by another process, as discussed in Section III-B. If so, an error number is returned to the caller. Otherwise the process can do one of the following two things. If the file is originally stored on a remote node, the file is transferred to the local node in the *fopen()* procedure, after which the process writes to the local copy. If the file to be written is right on the local node, or it is a new file, then the process starts writing the file just like a system call.

The aggregate write throughput is obviously optimal because file writes are associated with local I/O throughput and avoids the following two types of cost: (1) the procedure to determine to which node the data will be written, normally accomplished by pinging the metadata nodes or some monitoring services, and (2) transferring the data to a remote node. The downside of this file write strategy is the poor control on the load balance of compute node storage. This issue could be addressed by an asynchronous re-balance procedure running in the background, or by a load-aware task scheduler that steals tasks from the active nodes to the more idle ones.

When the process finishes writing to a file that is originally stored in another node, FusionFS does not send the newly modified file back to its original node. Instead, the metadata of this file is updated. This saves the cost of transferring the file data over the network.

D. File Read

Unlike file write, it is impossible to arbitrarily control where the requested data reside for file read. The location of the requested data is highly dependent on the I/O pattern. However, we could determine which node the job is executed on by the distributed workflow system, e.g. Swift [29]. That is, when a job on node A needs to read some data on node B, we reschedule the job on node B. The overhead of rescheduling the job is typically smaller than transferring the data over the network, especially for data-intensive applications. In our previous work [30], we detailed this approach, and justified it with theoretical analysis and experiments on benchmarks and real applications.

Indeed, remote readings are not always avoidable for some I/O patterns, e.g. merge sort. In merge sort, the data need to be joined together, and shifting the job cannot avoid the aggregation. In such cases, we need to transfer the requested data from the remote node to the requesting node. The data movement across compute nodes within FusionFS is conducted by the FDT service discussed in Section IV-A. FDT service is deployed on each compute node, and keeps listening to the incoming fetch and send requests.

E. File Close

Figure 5 shows the protocol when closing a file in FusionFS. In steps 1 – 3 the application on Node-i closes and flushes the

file to the local disk. If this is a read-only operation before the file is closed, then *libfuse* only needs to signal the caller (i.e. the application) in step 10. If this file has been modified, then its metadata needs to be updated in steps 4 – 7. Moreover, the replicas of this file also need to be updated in steps 8 – 9.

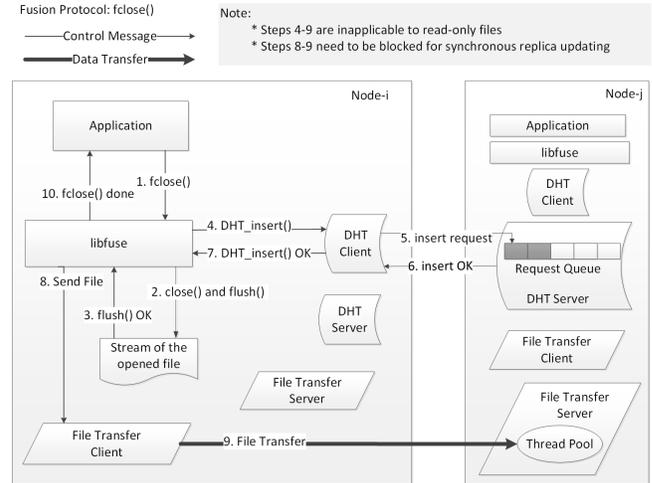


Figure 5. The protocol of file close in FusionFS

Again, just like Figure 4, the replica is not necessarily stored on the same node of its metadata (Node-j). Here we just show its remote replica on Node-j just to save some space of this paper.

V. EVALUATION

While we indeed compare FusionFS to some open-source systems such as PVFS [14] (in Figure 7) and HDFS [9] (in Figure 11), our top mission is to evaluate its performance improvement over the production file system of today's fastest systems. If we look at today's top 10 supercomputers [31], 4 systems are IBM Blue Gene/Q systems which run GPFS [12] as the default file system. Therefore most large-scale experiments conducted in this paper are carried out on Intrepid [13], a 40K-node IBM Blue Gene/P supercomputer whose default file system is also GPFS. Each Intrepid compute node has quad core 850MHz PowerPC 450 processors and runs a light-weight Linux ZeptoOS [32] with 2GB memory. A 7.6PB GPFS [12] parallel file system is deployed on 128 storage nodes. When FusionFS is evaluated as a POSIX-compliant file system, each compute node gets access to a local storage mount point with 174MB/s throughput on par with today's high-end hard drives. It points to the ramdisk and is throttled by a single-threaded FUSE layer. The network protocols for metadata management and file manipulation are TCP and FDT, respectively.

All experiments are repeated at least five times until results become stable (within 5% margin of error). The reported numbers are the average of all runs. Caching effect is carefully precluded by reading a file larger than the on-board memory before the measurement.

A. Metadata Rate

We expect that the metadata performance of FusionFS should be significantly higher than the remote GPFS on Intrepid, because FusionFS manipulates metadata in a completely distributed manner on compute nodes while GPFS has a limited number of clients on I/O nodes (every 64 compute nodes share one I/O node in GPFS). To quantitatively study the improvement, both FusionFS and GPFS create 10K empty files from each client on its own directory on Intrepid. That is, at 1024-nodes scale, we create 10M files over 1024 directories. We could have let all clients write on the same directory, but this workload would not take advantage of GPFS’ multiple I/O nodes. That is, we want to optimize GPFS’ performance when comparing it to FusionFS.

As shown in Figure 6, at 1024-nodes scale, FusionFS delivers nearly two orders of magnitude higher metadata rate over GPFS. FusionFS shows excellent scalability, with no sign of slowdown up to 1024-nodes. The gap between GPFS and FusionFS metadata performance would continue to grow, as eight nodes are enough to saturate the metadata servers of GPFS.

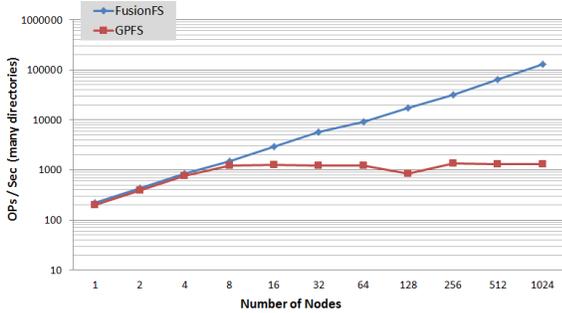


Figure 6. Metadata performance of FusionFS and GPFS on Intrepid (many directories)

One might overlook FusionFS’ novel metadata design and state that GPFS is slower than FusionFS simply because GPFS has fewer metadata servers (128) and fewer I/O nodes (1:64). First of all, that is the whole point why FusionFS is designed like this: to maximize the metadata concurrency without adding new resources to the system.

To really answer the question “what if a parallel file system has the same number of metadata servers just like FusionFS?”, we install PVFS [14] on Intrepid compute nodes with the 1-1-1 mapping between clients, metadata servers, and data servers just like FusionFS. We do not deploy GPFS on compute nodes because it is a proprietary system, and PVFS is open-source. The result is reported in Figure 7. Both FusionFS and PVFS turn on the POSIX interface with FUSE. Each client creates 10K empty files on the same directory to push more pressure on both systems’ metadata service. FusionFS outperforms PVFS even for a single client, which justifies that the metadata optimization for the big directory (i.e. update → append) on FusionFS is highly effective. Unsurprisingly, FusionFS again shows linear scalability. On the other hand, PVFS is saturated

at 32 nodes, suggesting that more metadata servers in parallel file systems do not necessarily improve the capability to handle higher concurrency.

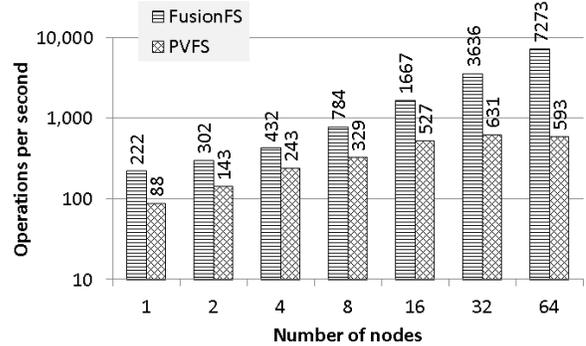


Figure 7. Metadata performance of FusionFS and PVFS on Intrepid (single directory)

B. I/O Throughput

Similarly to the metadata, we expect a significant improvement to the I/O throughput from FusionFS. Figure 8 shows the aggregate write throughput of FusionFS and GPFS on up to 1024-nodes of Intrepid. FusionFS shows almost linear scalability across all scales. GPFS scales at a 64-nodes step because every 64 compute nodes share one I/O node. Nevertheless, GPFS is still orders of magnitude slower than FusionFS at all scales.

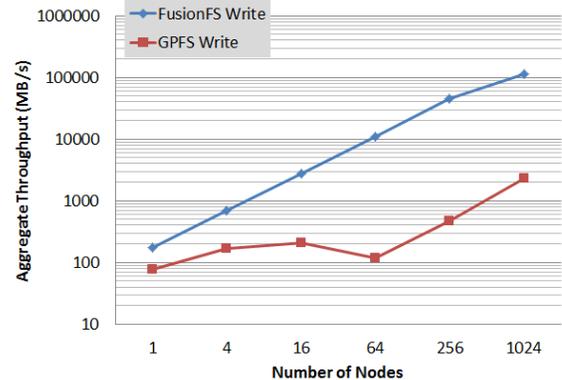


Figure 8. Write throughput of FusionFS and GPFS on Intrepid

Figure 9 shows FusionFS’ scalability at extreme scales. The experiment is carried out on Intrepid on up to 16K-nodes each of which has a FusionFS mount point. FusionFS throughput shows about linear scalability: doubling the number of nodes yield doubled throughput. Specifically, we observe stable 2.5TB/s throughput (peak 2.64TB/s) on 16K-nodes.

The main reason why FusionFS data write is faster is that the compute node only writes to its local storage. This is not true for data read though: it is possible that one node needs to transfer some remote data to its local disk. Thus, we are

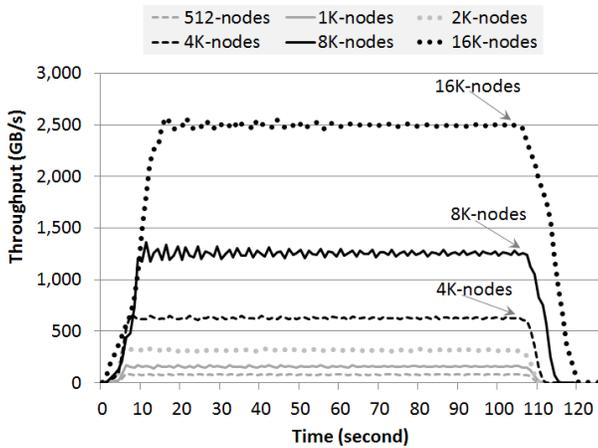


Figure 9. FusionFS scalability on Intrepid

interested in two extreme scenarios (i.e. all-local read and all-remote read) that define the lower and upper bounds of read throughput. We measure FusionFS for both cases on 256-nodes of Intrepid, where each compute node reads a file of different sizes from 1MB to 256MB. For the all-local case (e.g. where a data-aware scheduler can schedule tasks close to the data), all the files are read from the local nodes. For the all-remote case (e.g. where the scheduler is unaware of the data locality), every file is read from the next node in a round-robin fashion. This I/O pattern is unlikely realistic in real-world applications, but serves well as a workload for an all-remote request.

Figure 10 shows that FusionFS all-local read outperforms GPFS by more than one order of magnitude, as we have seen for data write. The all-remote read throughput of FusionFS is also significantly higher than GPFS, even though not as considerable as the all-local case. The reason why all-remote reads still outperforms GPFS is, again, FusionFS’ main concept of co-locating computation and data on the compute nodes: the bi-section bandwidth across the compute nodes (e.g. 3D-Torus) is higher than the interconnect between the compute nodes and the storage nodes (e.g. Ethernet fat-tree).

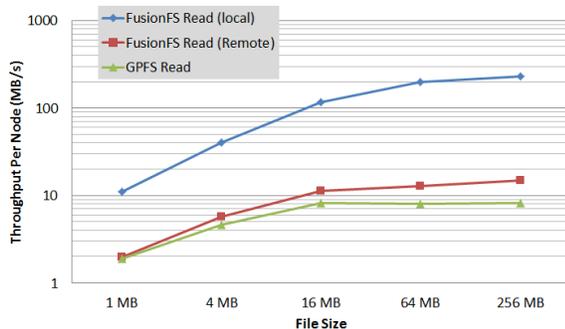


Figure 10. Read throughput of FusionFS and GPFS on Intrepid

In practice, the read throughput is somewhere between the two bounds, depending on the access pattern of the application and whether there is a data-aware scheduler to

optimize the task placement. FusionFS exposes this much needed data locality (via the metadata service) in order for parallel programming systems (e.g. Swift [29]) and job scheduling systems (e.g. Falkon [33]) to implement the data-aware scheduling. Note that Falkon has already implemented a data-aware scheduler for the “data diffusion” storage system [33], a precursor to the FusionFS project that lacked distributed metadata management, hierarchical directory-based namespace, and POSIX support. One potential improvement to FusionFS’ read throughput lies on better algorithms for predicting the future I/O patterns; we plan to explore this direction with incremental algorithms such as [34–36].

It might be argued that FusionFS outperforms GPFS mainly because FusionFS is a distributed file system on compute nodes, and the bandwidth is higher than the network between the compute nodes and the storage nodes. First of all, that is the whole point of FusionFS: taking advantage of the fast interconnects across the compute nodes. Nevertheless, we want to emphasize that FusionFS’ unique I/O strategy also plays a critical role in reaching the high and scalable throughput, as discussed in Section IV-C. So it would be a more fair game to compare FusionFS to other distributed file systems in the same hardware, architecture, and configuration. To show such a comparison, we deploy FusionFS and HDFS [9] on the Kodiak [37] cluster. We compare them on Kodiak because Intrepid does not support Java (required by HDFS).

Kodiak is a 1024-nodes cluster at Los Alamos National Laboratory. Each Kodiak node has an AMD Opteron 252 CPU (2.6GHz), 4GB RAM, and two 7200rpm 1TB hard drives. In this experiment, each client of FusionFS and HDFS writes 1GB data to the file system. Both file systems set replica to 1 to achieve the highest possible performance, and turn off the FUSE interface.

Figure 11 shows that the aggregate throughput of FusionFS outperforms HDFS by about an order of magnitude. FusionFS shows an excellent scalability, while HDFS starts to taper off at 256 nodes, mainly due to the weak write locality as data chunks (64MB) need to be scattered out to multiple remote nodes.

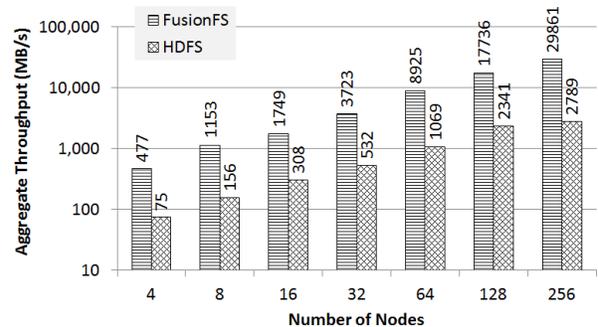


Figure 11. Throughput of FusionFS and HDFS on Kodiak

It should be clear that FusionFS is not to compete with HDFS, but to target the scientific applications on HPC machines that HDFS is not originally designed for or even

suitable for. So we have to restrict our design to fit for the typical HPC machine specification: a massive number of homogeneous and less-powerful cores with limited per-core RAM. Therefore for a fair comparison, when compared to FusionFS we had to deploy HDFS on the same hardware, which may or may not be an ideal or optimized testbed for HDFS.

C. Applications

We are interested in, quantitatively, how FusionFS helps to reduce the I/O cost for real applications. This section will evaluate four scientific applications on FusionFS all on Intrepid. The performance is mainly compared to Intrepid’s default storage, the GPFS [12] parallel file system.

For the first three applications, we replay the top three write-intensive applications on Intrepid [13] in December 2011 [3] on FusionFS: PlasmaPhysics, Turbulence, and AstroPhysics. While the PlasmaPhysics makes significant use of unique file(s) per node, the other two write to shared files. FusionFS is a file-level distributed file system, so PlasmaPhysics is a good example to benefit from FusionFS. However, FusionFS does not provide good N-to-1 write support for Turbulence and AstroPhysics. To make FusionFS’ results comparable to GPFS for Turbulence and AstroPhysics, we modify both workloads to write to unique files as the exclusive chunks of the share file. Due to limited space, only the first five hours of these applications running on GPFS are considered.

Figure 12 shows the real-time I/O throughput of these workloads at 1024-nodes. On FusionFS, these workloads are completed in 2.38, 4.97, and 3.08 hours, for PlasmaPhysics, Turbulence, and AstroPhysics, respectively. Recall that all of these workloads are completed in 5 hours in GPFS.

It is noteworthy that for both the PlasmaPhysics and AstroPhysics applications, the peak I/O rates for GPFS top at around 2GB/s while for FusionFS they reach over 100GB/s. This increase in I/O performance accelerates the applications 2.1X times (PlasmaPhysics) and 1.6X times (AstroPhysics). The reason why Turbulence does not benefit much from FusionFS is that, there are not many consecutive I/O operations in this application and GPFS is sufficient for such workload patterns: the heavy interleaving of I/O and computation does not push much I/O pressure to the storage system.

The fourth application, Basic Local Alignment Search Tool (BLAST), is a popular bioinformatics application to benchmark parallel and distributed systems. BLAST searches one or more nucleotide or protein sequences against a sequence database and calculates the similarities. It has been implemented with different parallelized frameworks, e.g. Parallel-BLAST [38]. In ParallelBLAST, the entire database (4GB) is split into smaller chunks on different nodes. Each node then formats its chunk into an encoded slice, and searches protein sequence against the slice. All the search results are merged together into the final matching result.

We compared ParallelBLAST performance on FusionFS and GPFS with our AME (Any-scale MTC Engine) framework [39]. We carried out a weak scaling experiment of

ParallelBLAST with 4GB database on every 64-nodes, and increased the database size proportionally to the number of nodes. The application has three stages (formatdb, blastp, and merge), which produces an overall data I/O of 541GB over 16192 files for every 64-nodes. In our experiment of 1024-node scale, the total I/O is about 9TB applied to over 250,000 files.

As shown in Figure 13, there is a huge (more than one order of magnitude) performance gap between FusionFS and GPFS at all scales, except for the trivial 1-node case. FusionFS has up to 32X speedup (at 512-nodes), and an average of 23X improvement between 64-nodes and 1024-nodes. At 1-node scale, the GPFS kernel module is more effective in accessing an idle parallel file system. In FusionFS’ case, the 1-node scale result involves the user-level FUSE module, which apparently causes BLAST to run 1.4X slower on FusionFS. However, beyond the corner-case of 1-node, FusionFS significantly outperforms GPFS. In particular, on 1024-nodes BLAST requires 1,073 seconds to complete all three stages on FusionFS, and it needs 32,440 seconds to complete the same workload on GPFS.

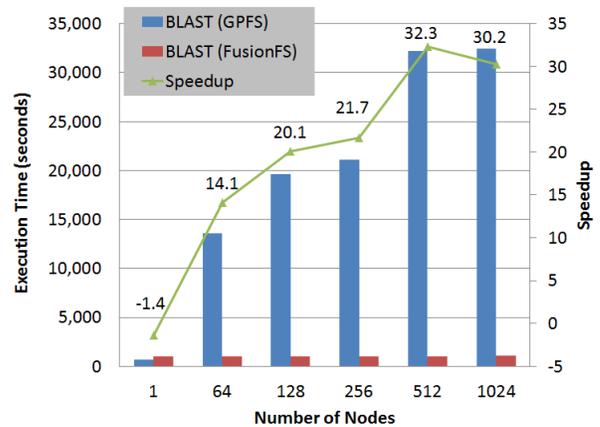


Figure 13. BLAST execution time on Intrepid

VI. RELATED WORK

There have been many shared and parallel file systems, such as the Network File System (NFS [40]), General Purpose File System (GPFS [12]), Parallel Virtual File System (PVFS [14]), Lustre[18], and Panasas[11]. These systems assume that storage nodes are significantly fewer than the compute nodes, and compute resources are agnostic of the data locality on the underlying storage system, which results in an unbalanced architecture for data-intensive workloads.

A variety of distributed file systems have been developed such as Google File System (GFS [26]), Hadoop File System (HDFS [9]), Ceph [41], and Sector [42]. However, many of these file systems are tightly coupled with execution frameworks (e.g. MapReduce [43]), which means that scientific applications not using these frameworks must be modified to use these non-POSIX file systems. For those that offer a POSIX interface, they are not designed for metadata-intensive

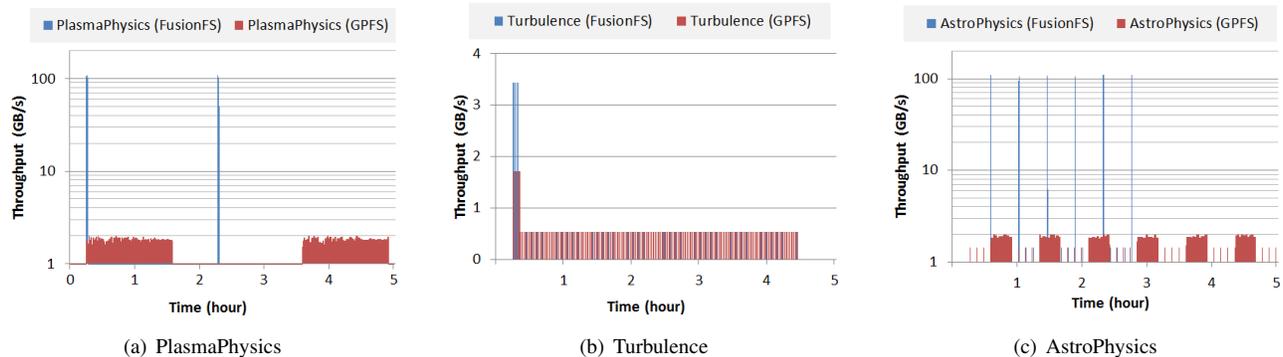


Figure 12. Top three write-intensive applications on Intrepid

operations at extreme scales. The majority of these systems do not expose the data locality information for general computational frameworks (e.g. batch schedulers, workflow systems) to harness the data locality through data-aware scheduling. In short, these distributed file systems are not designed specifically for HPC and scientific computing workloads, and the scales that HPC are anticipating in the coming years.

The idea of distributed metadata can be traced back to xFS [44], even though a central manager is in need to locate a particular file. Recently, FDS [45] was proposed as a blob store on data centers. It maintains a lightweight metadata server and offloads the metadata to available nodes in a distributed manner. In contrast, FusionFS metadata is completely distributed without any single-point-of-failure involved.

Co-location of compute and storage resources has attracted a lot of research interests. For instance, Salus [46] proposes to co-locate the storage to data nodes in data centers. Other examples include Rhea [47], which prevents removing the data used by the computation, and Nectar [48], which automatically manages data and computation in data centers. While these systems apply a general rule to deal with data I/O, FusionFS is optimized for write-intensive workloads that are particularly important for HPC systems.

VII. CONCLUSION AND FUTURE WORK

This paper proposes a distributed storage layer on compute nodes to tackle the HPC I/O bottleneck of scientific applications. We identify the challenges this unprecedented architecture brings, and build a distributed file system FusionFS to tackle them. In particular, FusionFS is crafted to support extremely intensive metadata operations and is optimized for file writes. Extreme-scale evaluation on up to 16K nodes demonstrates FusionFS' superiority over other popular storage systems for scientific applications. We plan to explore the feasibility to integrate a memory-centric middleware (e.g. Tachyon [49]) for cooperative caching [50].

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards OCI-1054974 (CAREER). This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported

by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research was supported by the United States Department of Defense. This work is also supported by the Department of Energy (DOE) Office of Advanced Scientific Computer Research (ASCR) under contract DE-AC02-06CH11357. This material is supported in part by the National Science Foundation under awards CNS-1042537 and CNS-1042543 (PRObE) [51]. <http://www.nmc-probe.org/>

REFERENCES

- [1] P. Freeman, D. Crawford, S. Kim, and J. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," *Proceedings of the IEEE*, vol. 93, no. 3, 2005.
- [2] D. Zhao, D. Zhang, K. Wang, and I. Raicu, "Exploring reliability of exascale systems through simulations," in *Proceedings of the 21st ACM/SCS High Performance Computing Symposium (HPC)*, 2013.
- [3] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, 2012.
- [4] W. Tantisirirotj, S. W. Son, S. Patil, S. J. Lang, G. Gibson, and R. B. Ross, "On the duality of data-intensive file system design: Reconciling HDFS and PVFS," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.
- [5] D. Zhao and I. Raicu, "Distributed file systems for exascale computing," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12), doctoral showcase*, 2012.
- [6] DEEP-ER, "<http://www.hpc.cineca.it/projects/deep-er>," 2014.
- [7] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to efficiently leverage multicore parallelism to achieve scalable, jitter-free I/O," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, 2012.
- [8] Mira, "<https://www.alcf.anl.gov/user-guides/mira-cetus-vesta>," 2014.
- [9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, 2010.
- [10] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Mass Storage Systems and Technologies, 2013 IEEE 29th Symposium on*, 2013.
- [11] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas activescale storage cluster: Delivering scalable high bandwidth

- storage,” in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, 2004.
- [12] F. Schmuck and R. Haskin, “GPFS: A shared-disk file system for large computing clusters,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST)*, 2002.
 - [13] Intrepid, “<https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor>,” 2014.
 - [14] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, “PVFS: A parallel file system for linux clusters,” in *Proceedings of the 4th Annual Linux Showcase and Conference*, 2000.
 - [15] D. Zhao, J. Yin, K. Qiao, and I. Raicu, “Virtual chunks: On supporting random accesses to scientific data in compressible storage systems,” in *Proceedings of the 2014 IEEE International Conference on Big Data (IEEE BigData)*, 2014.
 - [16] D. Zhao, Y. Jian, and I. Raicu, “Improving the i/o throughput for data-intensive scientific applications with efficient compression mechanisms,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13), poster session*, 2013.
 - [17] FUSE Project, “<http://fuse.sourceforge.net>,” 2014.
 - [18] P. Schwan, “Lustre: Building a file system for 1,000-node clusters,” in *Proceedings of the linux symposium*, 2003.
 - [19] A. Rajgarhia and A. Gehani, “Performance and extension of user space file systems,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010.
 - [20] D. Zhao and I. Raicu, “HyCache: A user-level caching middleware for distributed file systems,” in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW)*, 2013.
 - [21] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, “ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table,” in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.
 - [22] D. Zhao, C. Shou, T. Malik, and I. Raicu, “Distributed data provenance for large-scale data-intensive computing,” in *Cluster Computing (CLUSTER), 2013 IEEE International Conference on*, 2013.
 - [23] C. Shou, D. Zhao, T. Malik, and I. Raicu, “Towards a provenance-aware distributed filesystem,” in *5th Workshop on the Theory and Practice of Provenance (TAPP)*, 2013.
 - [24] D. Zhao, K. Burlingame, C. Debains, P. Alvarez-Tabio, and I. Raicu, “Towards high-performance and cost-effective distributed storage systems with information dispersal algorithms,” in *Cluster Computing, 2013 IEEE International Conference on*, 2013.
 - [25] Protocol Buffers, “<http://code.google.com/p/protobuf/>,” 2014.
 - [26] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.
 - [27] epoll, “<http://man7.org/linux/man-pages/man7/epoll.7.html>,” 2014.
 - [28] Y. Gu and R. L. Grossman, “Supporting configurable congestion control in data transport services,” in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.
 - [29] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, “Swift: Fast, reliable, loosely coupled parallel computation,” in *Proceedings of the 2007 IEEE Congress on Services*, 2007.
 - [30] I. Raicu, I. T. Foster, Y. Zhao, P. Little, C. M. Moretti, A. Chaudhary, and D. Thain, “The quest for scalable support of data-intensive workloads in distributed systems,” in *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC)*, 2009.
 - [31] Top500, “<http://www.top500.org/list/2014/06/>,” 2014.
 - [32] ZeptoOS, “<http://www.mcs.anl.gov/zeptoos>,” 2014.
 - [33] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, “Falkon: a fast and light-weight task execution framework,” in *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.
 - [34] D. Zhao and L. Yang, “Incremental isometric embedding of high-dimensional data using connected neighborhood graphs,” *IEEE Trans. Pattern Anal. Mach. Intell. (PAMI)*, vol. 31, no. 1, Jan. 2009.
 - [35] R. Lohfert, J. Lu, and D. Zhao, “Solving sql constraints by incremental translation to sat,” in *21st International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2008.
 - [36] D. Zhao and L. Yang, “Incremental construction of neighborhood graphs for nonlinear dimensionality reduction,” in *Proceedings of the 18th International Conference on Pattern Recognition - Volume 03*, 2006.
 - [37] Kodiak, “<https://www.nmc-probe.org/wiki/kodiak:nodes>,” 2014.
 - [38] D. R. Mathog, “Parallel BLAST on split databases,” *Bioinformatics*, vol. 19(4), pp. 1865 – 1866, 2003.
 - [39] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. T. Foster, “Design and analysis of data management in scalable parallel scripting,” in *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, 2012.
 - [40] M. Eisler, R. Labiaga, and H. Stern, “Managing NFS and NIS, 2nd ed.” *O’Reilly & Associates, Inc.*, 2001.
 - [41] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, “Ceph: A scalable, high-performance distributed file system,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.
 - [42] Y. Gu, R. L. Grossman, A. Szalay, and A. Thakar, “Distributing the Sloan Digital Sky Survey using UDT and Sector,” in *Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, 2006.
 - [43] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation (OSDI)*, 2004.
 - [44] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, “Serverless network file systems,” in *Proceedings of the fifteenth ACM symposium on Operating systems principles (SOSP)*, 1995.
 - [45] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, “Flat datacenter storage,” in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2012.
 - [46] Y. Wang, M. Kapritsos, Z. Ren, P. Mahajan, J. Kirubanandam, L. Alvisi, and M. Dahlin, “Robustness in the salus scalable block store,” in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.
 - [47] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron, “Rhea: automatic filtering for unstructured cloud storage,” in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, 2013.
 - [48] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, “Nectar: automatic management of data and computation in datacenters,” in *Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI)*, 2010.
 - [49] Tachyon, “<http://tachyon-project.org/>,” 2014.
 - [50] D. Zhao, K. Qiao, and I. Raicu, “Hycache+: Towards scalable high-performance caching middleware for parallel file systems,” in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2014.
 - [51] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd, “Probe: A thousand-node experimental cluster for computer systems research,” vol. 38, no. 3, June 2013.

The submitted manuscript has been created by UChicago Argonne, LLC, Operator of Argonne National Laboratory ("Argonne"). Argonne, a U.S. Department of Energy Office of Science laboratory, is operated under Contract No. DE-AC02-06CH11357. The U.S. Government retains for itself, and others acting on its behalf, a paid-up nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.