Towards Exploring Data-Intensive Scientific Applications at Extreme Scales through Systems and Simulations

Dongfang Zhao, Ning Liu, Dries Kimpe, Robert Ross, Xian-He Sun, and Ioan Raicu

Abstract—The state-of-the-art storage architecture of high-performance computing systems was designed decades ago, and with today's scale and level of concurrency, it is showing significant limitations. Our recent work proposed a new architecture to address the I/O bottleneck of the conventional wisdom, and the system prototype (FusionFS) demonstrated its effectiveness on up to 16 K nodes—the scale on par with today's largest supercomputers. The main objective of this paper is to investigate FusionFS's scalability towards exascale. Exascale computers are predicted to emerge by 2018, comprising millions of cores and billions of threads. We built an event-driven simulator (FusionSim) according to the FusionFS architecture, and validated it with FusionFS's traces. FusionSim introduced less than 4 percent error between its simulation results and FusionFS traces. With FusionSim we simulated workloads on up to two million nodes and find out almost linear scalability of I/O performance; results justified FusionFS's viability for exascale systems. In addition to the simulation work, this paper extends the FusionFS system prototype in the following perspectives: (1) the fault tolerance of file metadata is supported, (2) the limitations of the current system design is discussed, and (3) a more thorough performance evaluation is conducted, such as N-to-1 metadata write, system efficiency, and more platforms such as Amazon Cloud.

Index Terms—Data storage systems, file systems, high performance computing, supercomputers

1 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, occasional checkpoints, and the final output. Therefore, since the bottleneck used to be the computation, significantly more resources have been invested in the computational capabilities of these systems.

In the era of Big Data, however, scientific applications are becoming data-centric [1]; their workloads are now data-intensive rather than compute-intensive, requiring a greater degree of support from the storage subsystem [2]. Making it worse, the existing gap between compute and I/O continues to widen as the growth of compute system still follows Moore's Law while the growth of storage systems has severely lagged behind. Prior work [3] shows that current HPC storage architecture would not scale to the emerging exascale computing systems (10^{18} ops/s).

While recent studies (for example, [4], [5]) address the I/O bottleneck in the conventional architecture of HPC systems, this work is orthogonal to them by proposing a new HPC architecture that collocates node-local storage with compute resources. In particular, we envision a distributed storage system on compute nodes for applications to manipulate their intermediate results and checkpoints; the data only need to be transferred over the network to the remote storage for archival purposes. While co-location of storage and computation has been widely adopted in cloud computing and data centers (for example, Hadoop clusters [6]) and extensively studied in in-situ HPC data analysis [7], [8], [9], [10], [11], a real system having persistent storage deployed on compute nodes of a HPC system has never existed, despite this architecture attracting much research interest (for example, DEEP-ER [12]). This work, to the best of our knowledge, for the first time 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 may raise concerns about jitters on compute nodes, since applications' computation and I/O would share resources such as CPU cycles and network bandwidth. Nevertheless, recent study [13] shows that the I/O-related cost can be offloaded onto dedicated infrastructures that are decoupled from the application's acquired resources, making computation and I/O separated at a finer granularity. In fact, this resource-isolation strategy has been applied in production systems: the IBM Blue Gene/Q supercomputer (Mira [14])

- *D. Zhao and N. Liu are with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616. E-mail: {dzhao8, nliu8}@iit.edu.*
- *D. Kimpe and R. Ross are with Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. E-mail: {dkimpe, rross}@mcs.anl.gov.*
- *X. Sun and I. Raicu are with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616, and the Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. E-mail: {sun, iraicu}@cs.iit.edu.*

Manuscript received 11 Nov. 2014; revised 8 May 2015; accepted 29 June 2015. Date of publication 14 July 2015; date of current version 18 May 2016. Recommended for acceptance by A. R. Butt.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2456896

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 cloud computing and data centers (for example, the popular distributed file system HDFS [15]); yet there exists little literature for building a distributed storage system particularly for HPC systems whose design principles are greatly different from data centers. As a case in point, HPC nodes are highly customized and tightly coupled with high-throughput and low-latency network (for example, InfiniBand), while data centers typically have commodity servers and inexpensive networks (for example, Ethernet). Therefore, storage systems optimized for data centers might not be appropriate for HPC systems, as we will see HDFS delivers poor performance on a typical HPC machine in Fig. 9. Specifically, we observe that the following challenges are unique to a distributed storage on HPC systems, related to both metadata-intensive and write-intensive workloads.

First, the storage system on HPC nodes needs to support intensive metadata operations. Many scientific applications create a large number of small- to medium-sized files, as Welch and Noer [16] report that 25-90 percent of all the 600 million files from 65 Panasas [17] installations are 64 KB or smaller. This implies that the I/O performance is highly throttled by the metadata rate, besides the data itself. Data centers do not optimize for this type of workload, as claimed in Google file system (GFS) [18]: "Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them." Indeed, if we recall that Hadoop distributed file system, [15] (an open-source variation of GFS) splits a large file into a series of default 64 MB chunks (128 MB recommended in most cases) for parallel processing, a small- or medium-sized file (for example, 64 KB) can benefit little from this data parallelism. Moreover, the centralized metadata server in GFS and HDFS is obviously not designed to handle intensive metadata operations. In the HPC community, traditional parallel file systems (for example, IBM General Parallel File System, GPFS [19], Parallel Virtual File System, PVFS [20]) that could claim to have some non-centralized metadata management, in fact have a limited form of distribution that often limits the aggregate performance of metadata operations significantly [21], for instance see Fig. 3. Therefore, this work argues that the metadata management in HPC storage systems needs to be revisited.

Second, file writes should be optimized for the distributed storage on HPC nodes. The fault tolerance of most 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. In this checkpoint-restart scenario, file writes usually outnumber file reads in terms of both frequency and size; improving the write performance would 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. Therefore, conventional wisdom and experience from data center storage could not be directly employed by HPC storage systems.

We implement the fusion distributed file system (FusionFS) to demonstrate how to overcome the aforementioned challenges. FusionFS disperses its metadata to all the available nodes to achieve the maximal concurrency of metadata operations. Every client of FusionFS optimizes file writes with local writes (whenever possible), which reduces network traffic and makes the aggregate I/O throughput highly scalable (almost linear, see Fig. 17). FusionFS is supposed to coexist with the remote storage (for example, GPFS [19]) rather than to replace the latter, which mainly serves as the archival repository.

FusionFS is recently deployed on up to 16 K nodes of an IBM Blue Gene/P supercomputer (Intrepid [22]), and heavily accessed by a variety of benchmarks and applications. We observe more than an order of magnitude improvement to the I/O performance when comparing FusionFS to other popular file systems such as GPFS [19], PVFS [20], and HDFS [15], surpassing 2.5 TB/s aggregate I/O throughput on 16 K nodes. To understand FusionFS's scalability beyond today's largest systems, an event-driven simulator, FusionSim, is developed and validated by FusionFS traces. FusionSim simulation shows that FusionFS scales almost linearly and delivers 329 TB/s on 2-million nodes, the scale many experts believe represents the exascale computing system.

Preliminary results of FusionFS are recently published in the IEEE International Conference on Big Data. In [23], we propose the new storage architecture for extreme-scale HPC systems to address the I/O bottleneck of modern data-intensive scientific applications. Based on the new architecture, we design and implement the FusionFS file system to support metadata- and write-intensive workloads. Extensive evaluation is conducted with benchmarks and applications at extreme scales; results demonstrated FusionFS's superiority over state-of-the-art solutions (for example, GPFS [19], PVFS [20], and HDFS [15]). This paper significantly extends our prior work [23] in the following perspectives:

- *Deeper study of distributed metadata management, in both performance and resilience*
- *Extended performance evaluation of FusionFS with additional file systems (for example, PVFS [20]) in new environments (for example, Amazon Cloud)*
- *A simulation study to predict FusionFS's performance beyond today's largest systems towards exascales, which is validated by FusionFS traces*

The remainder of this paper is structured as follows. Section 2 presents the design and implementation of the FusionFS file system. Section 3 describes the design and implementation of the FusionSim simulator. We report experimental results in Section 4. Section 5 reviews related work on HPC storage systems. We finally conclude this paper in Section 6.

2 The FusionFS File System

This section discusses the design trade-off of the FusionFS file system. More detail of system implementation was elaborated in [23].

As shown in Fig. 1, FusionFS is a user-level file system that runs on the compute resource infrastructure, and

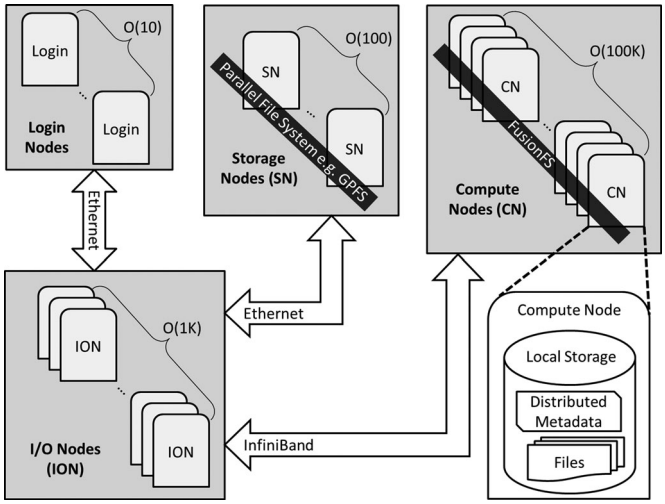


Fig. 1. FusionFS deployment in a typical HPC system.

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.

The POSIX interface is implemented with the FUSE framework [24], so that legacy applications can run directly on FusionFS without modifications. Nevertheless, 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 (*/dev/fuse*). A detailed comparison between FUSE-enabled and native file systems was reported in [25], showing that a Java implementation of a FUSE-based file system introduces about 60 percent 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 [26], we reported that FUSE could deliver as high as 578 MB/s throughput, 85 percent 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.

2.1 Metadata Management

2.1.1 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 [27], [28], [29], [30]), which disperses partial metadata on each compute node. The client could interact with the DHT to inquire about any file on any node. 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.

2.1.2 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 such as "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.

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.

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 [30] 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 [18], 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.

2.1.3 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 [31] 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.

2.1.4 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.

2.1.5 Consistency

Since each primary metadata copy has replicas, the next question is how to 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 (for example, last modification time), the first replica is strongly consistent to the primary copy, and the other replicas are updated asynchronously. By doing this, the metadata are strongly consistent (in the average case) while the overhead is kept relatively low.

2.1.6 Fault Tolerance

When a node fails, we need to restore the missing metadata and files on that node as soon as possible. The traditional method for data replication is to make a number of replicas to the primary copy. This method has its advantages such as ease-of-use, less compute-intensive, when compared to the emerging erasure-coding mechanism [32], [33]. The main critique on replicas is, however, its low storage efficiency. For example, in Google file system [18] each primary copy has two replicas, which results in the storage utilization as $\frac{1}{1+2} = 33\%$. For fault tolerance of metadata, we choose data replication for metadata based on the following observations. First, metadata size is typically much smaller than file data in orders of magnitude. Therefore, replicating the metadata impact little to the overall space utilization of the entire system. Second, the computation overhead introduced by erasure coding can hardly be amortized by the reduced I/O time on transferring the encoded metadata.

2.2 File Manipulation

2.2.1 File Movement

For file movement (i.e., data transfer over the network across compute nodes), 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) [34]. 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.

2.2.2 File Open

When the application on machine A issues a POSIX fopen() call, it is caught by the implementation in the FUSE user-level interface (i.e., *libfuse*) for file open. The metadata client then retrieves the file location from the metadata server that is implemented by a distributed hash table [30]. The location information might be stored in another machine B, so this procedure could involve a round trip of messages between A and B. Then A needs to ping B to fetch the file. Finally the local operating system triggers the system call to open the transferred file and finally returns the file handle to the application.

2.2.3 File Write

Before writing to a file, the process checks if the file is being accessed by another process, as discussed in Section 2.1.2. 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. It should be clear that FusionFS works at the file level, thus chunking the file is not an option. Nevertheless, we will support chunk-level data movement in the next release of FusionFS, whose preliminary results are currently under review [35]. 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.

2.2.4 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 such as

Swift [36]. 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 [37], 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 such as 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 2.2.1. FDT service is deployed on each compute node, and keeps listening to the incoming fetch and send requests.

2.2.5 File Close

When the application on Node-i closes, it 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). If this file has been modified, then its metadata needs to be updated. Moreover, the replicas of this file also need to be updated.

Again, just like file open, the replica is not necessarily stored on the same node of its metadata (Node-j). Here we just assume its remote replica on Node-j for clear presentation.

3 The FusionSim Simulator

This section presents the design and implementation of the FusionSim simulator, aimed to simulate FusionFS's performance at the scales beyond today's largest systems. We will first introduce two building blocks of FusionSim—ROSS and CODES, and then move to the FusionSim simulator.

3.1 Building Blocks

3.1.1 The ROSS Simulation Tool

ROSS [38] is a massively parallel discrete-event simulation (PDES) system developed in Rensselaer Polytechnic Institute. It uses the time warp algorithm and features reverse computation for optimistic simulation. Users can choose to build and run the models in sequential, conservative or optimistic mode. To date, researchers have built many successful large-scale models using ROSS. In [39], ROSS has demonstrated the ability to process billions of events per second by leveraging large-scale HPC systems.

A parallel discrete-event simulation system consists of a collection of logical processes (LPs) that are used to model distinct components of the system (for example, a file server in FusionSim). LPs communicate by exchanging time stamped event messages (for example, denoting the arrival or departure of a request message). The goal of PDES is to efficiently process all events in a global timestamp order at the minimum overhead of any processor synchronization.

3.1.2 The CODES Simulation System

CODES [40] is a simulation system built on ROSS. Initially, the goal of CODES is to enable the exploration and co-design of exascale storage systems by providing a detailed, accurate, and highly parallel simulation toolkit for exascale storage systems. As CODES evolves, many modules emerge and have greatly enlarged the original scope. CODES has gradually become a comprehensive platform that supports the modeling and simulation of large-scale complex systems including operating systems, file systems, HPC systems, HPC applications, and data centers.

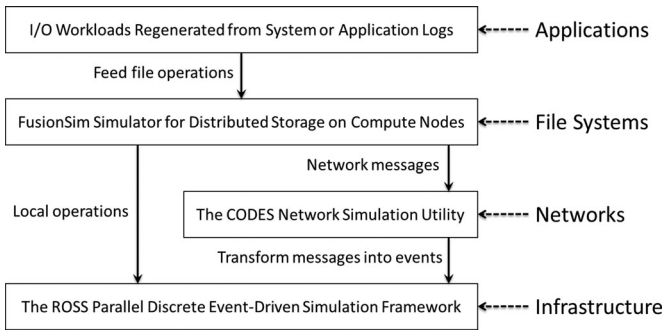


Fig. 2. The software stack of FusionSim.

To date, CODES comprises of the following modules: CODES-net, CODES-workloads, CODES-bg and CODES-lsm. Specifically, CODES-net provides four networking models based on parallel discrete-event simulation: the torus network model [41], the dragonfly network model [42], the LogGP model [43] and the simple-net model (a simple N-to-N network).

CODES-net provides unified interfaces that facilitate the use of all underlying networking models. FusionSim leverages the functionality provided by CODES torus network model and thus provides a detailed HPC communication model and simulation. We provide the details of FusionSim in the following section.

3.2 FusionSim

With the two enabling techniques, ROSS and CODES, we build FusionSim to simulate FusionFS behavior on the scales beyond today's largest systems. The full software stack is shown in Fig. 2. Four layers comprise the entire hierarchy of the simulation system (from top downwards): application, file systems, networks, and infrastructure.

When the simulation starts, the system first calls the application layer to generate the workload (i.e., file operations) as the input of the FusionFS file system. Then FusionSim at the file system layer decides where to deal with the file operation depending on its locality. In essence, if this file operation involves network transfer, it is redirected to the network models implemented in the CODES framework at the network layer. Otherwise, the file operation (i.e., an event) is passed to the ROSS infrastructure. It should be noted that the events at the network layer are eventually passed to the ROSS infrastructure as well.

FusionFS's logic is implemented at the file system layer in Fig. 2. Basically, each node in FusionFS is abstracted as a logical process in FusionSim. For any file operation (regardless it is related to metadata or data), it triggers a new event in the simulation system. The network topology is implemented at the network layer. For example we can specify the network type for the system to simulate; in this paper it is set to 3-D torus for the IBM Blue Gene/P supercomputer.

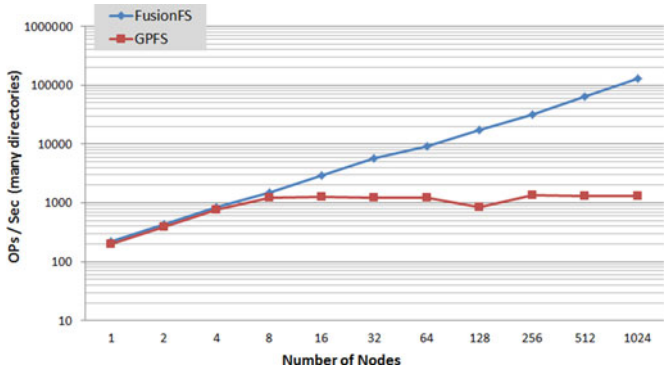


Fig. 3. Metadata performance of FusionFS and GPFS on Intrepid (many directories).

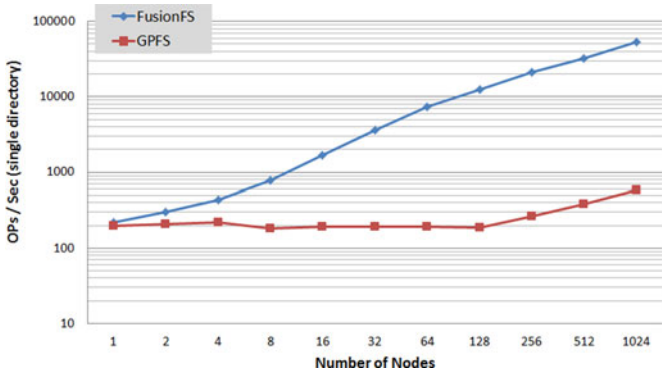


Fig. 4. Metadata performance of FusionFS and GPFS on Intrepid (single directory).

4 Evaluation

This section answers the following four questions:

1) How does FusionFS perform under intensive metadata workloads at large scale when comparing with other major file systems? (Section 4.2)
2) What is FusionFS's I/O throughput—not only (optimized) file writes but also file reads? (Section 4.3)
3) What performance improvement can real-world applications achieve from FusionFS? (Section 4.4)
4) How does (or, will) FusionFS scale at extreme-scale (for example, exascale) systems? (Section 4.5)

We will first describe the experiment setup for these evaluations, then report and discuss the results one after another.

4.1 Experiment Setup

While we indeed compare FusionFS to some open-source systems such as PVFS [20] (in Fig. 5) and HDFS [15] (in Fig. 9), 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 [44], four systems are IBM Blue Gene/Q systems which run GPFS [19] as the default file system. GPFS is deployed on 128 network-attached storage nodes; every 64 compute nodes share one I/O node to talk to the storage nodes. GPFS is setup like this by considering the tightly-coupled computing architecture and the I/O workload from these compute nodes. Most large-scale experiments conducted in this paper are carried out on Intrepid [22], a 40 K-node IBM Blue Gene/P supercomputer whose default file system is also GPFS.

Each Intrepid compute node has quad core 850 MHz PowerPC 450 processors and runs a light-weight Linux ZeptoOS [45] with 2 GB memory. A 7.6 PB GPFS [19] 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 174 MB/s throughput on par with today's high-end hard drives; this local disk is emulated by the ramdisk 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 percent 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. Except for Fig. 9, all experiments of FusionFS are carried out with the POSIX interface; That is, the FUSE overhead is included in the reported results.

4.2 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 10 K empty files from each client on its own directory on Intrepid. That is, at 1,024-nodes scale, we create 10 M files over 1,024 directories. This workload optimizes GPFS' performance by taking advantage of GPFS' multiple I/O nodes.

As shown in Fig. 3, at 1,024-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 1,024-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.

Each process only working on its own directory is rare in the real world. Therefore we are interested in another extreme case where many nodes create files in the same directory. We expect that GPFS performs significantly worse than the case of many directories. To confirm this, Fig. 4 shows the metadata throughput when multiple nodes create files in a single global directory. We observe that GPFS does not scale even with two nodes. In contrast, FusionFS delivers scalable throughput with similar trends as in the many-directory case. Of note, FusionFS achieves 54 K file creations per second on a single directory at 1 K-node scale, mainly due to the use of the append operation in DHT [30] allowing lock-free concurrent metadata writes on metadata objects.

One might overlook FusionFS's 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

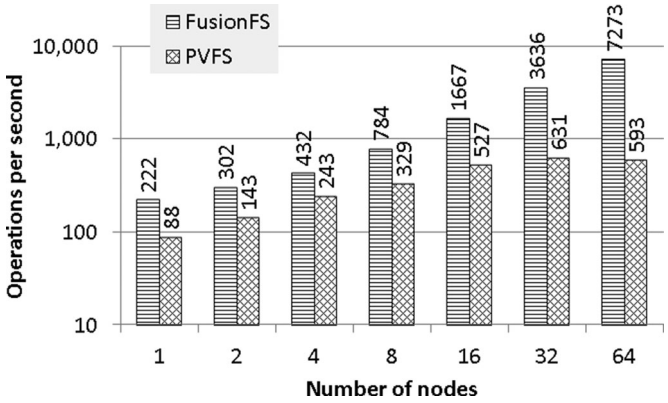


Fig. 5. Metadata performance of FusionFS and PVFS on Intrepid (single directory).

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 [20] 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 Fig. 5. Both FusionFS and PVFS turn on the POSIX interface with FUSE. Each client creates 10 K 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.

Lastly, we show the overhead introduced by metadata replication [30]. Fig. 6 shows that the overhead of adding two replicas is roughly 30 percent. It is relatively small because the replica is updated asynchronously.

4.3 I/O Throughput

Similarly to the metadata, we expect a significant improvement to the I/O throughput from FusionFS. Fig. 7 shows the aggregate write throughput of FusionFS and GPFS on up to 1,024-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.

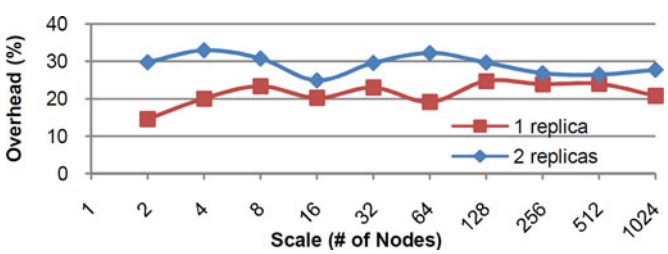


Fig. 6. Overhead of metadata replication.

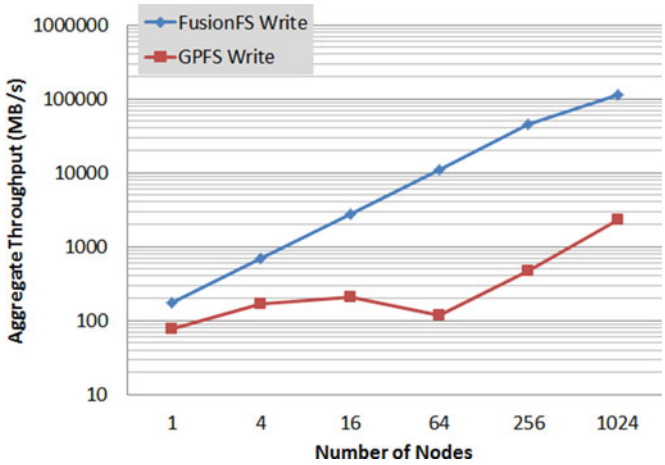


Fig. 7. Write throughput of FusionFS and GPFS on Intrepid.

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 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 1 to 256 MB. For the all-local case (for example, 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 (for example, 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 to be realistic in real-world applications, but serves well as a workload for an all-remote request.

Fig. 8 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's main concept of co-locating computation and data on the compute nodes: the bi-section bandwidth across the compute nodes (for example, 3D-Torus) is higher than the interconnect between the compute nodes and the storage nodes (for example, Ethernet fat-tree).

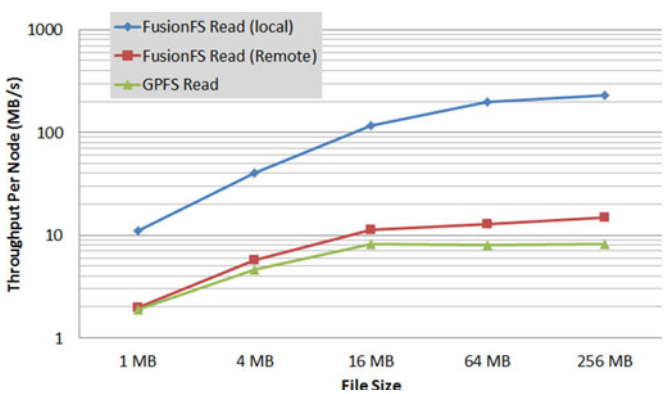


Fig. 8. Read throughput of FusionFS and GPFS on Intrepid.

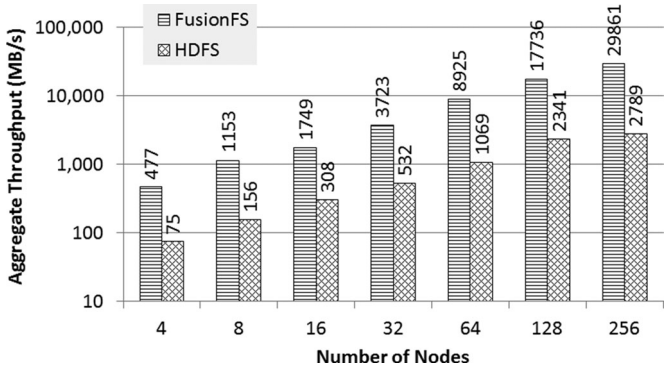


Fig. 9. Throughput of FusionFS and HDFS on Kodiak.

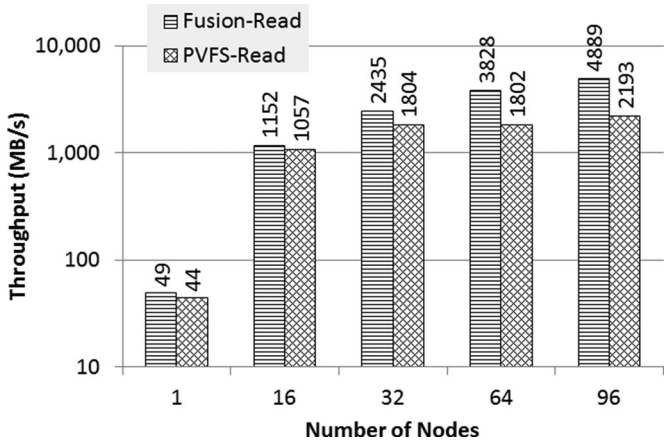


Fig. 10. Read throughput on Amazon EC2 Cloud.

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 (for example, Swift [36]) and job scheduling systems (for example, Falkon [46]) to implement the data-aware scheduling. Note that Falkon has already implemented a data-aware scheduler for the "data diffusion" storage system [46], a precursor to the FusionFS project that lacked distributed metadata management, hierarchical directory-based namespace, and POSIX support.

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's unique I/O strategy also plays a critical role in reaching the high and scalable throughput, as discussed in Section 2.2.3. 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 [15] on the Kodiak [47] cluster. We compare them on Kodiak because Intrepid does not support Java (required by HDFS).

Kodiak is a 1,024-nodes cluster at Los Alamos National Laboratory. Each Kodiak node has an AMD Opteron 252 CPU (2.6 GHz), 4 GB RAM, and two 7200 rpm 1 TB hard drives. In this experiment, each client of FusionFS and HDFS writes 1 GB data to the file system. Both file systems set replica to 1 to achieve the highest possible performance, and turn off the FUSE interface.

Fig. 9 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 (64 MB) need to be scattered out to multiple remote nodes.

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.

Similarly to the metadata evaluation, at the end of this section we compare FusionFS and the parallel filesystem with a similar architecture—PVFS. We compare both systems on Amazon EC2 with FUSE enabled. We use up to 96 m1.medium instances [48] in our experiments. Each instance has a single virtual core, 3.75 GB memory, and 410 GB storage space. Results are reported in Figs. 10 and 11. We observe that on a single node, the performance is comparable for both systems. However, FusionFS shows a better scalability and the gap between both is getting larger on more nodes. At 96-instance, FusionFS delivers a 2.2× higher read throughput and 3.6× higher write throughput.

4.4 Real-World 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 [19] parallel file system.

For the first three applications, we replay the top three write-intensive applications on Intrepid [22] in December 2011 [4] on FusionFS: PlasmaPhysics, Turbulence, and AstroPhysics. While the PlasmaPhysics makes significant use of unique file(s) per node, the other two write to shared

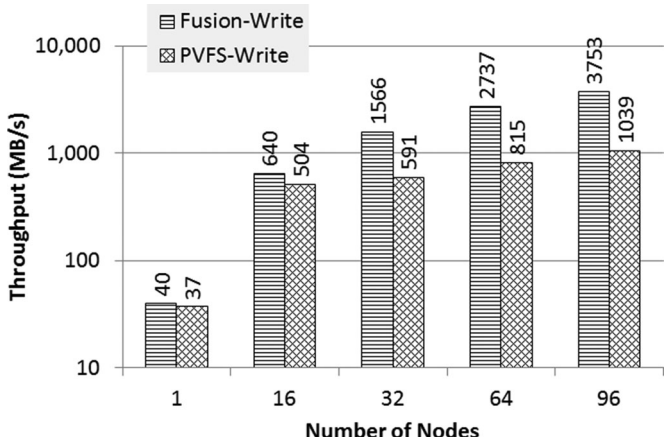
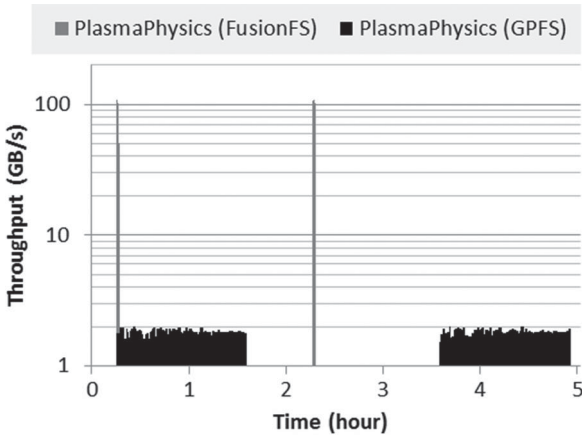
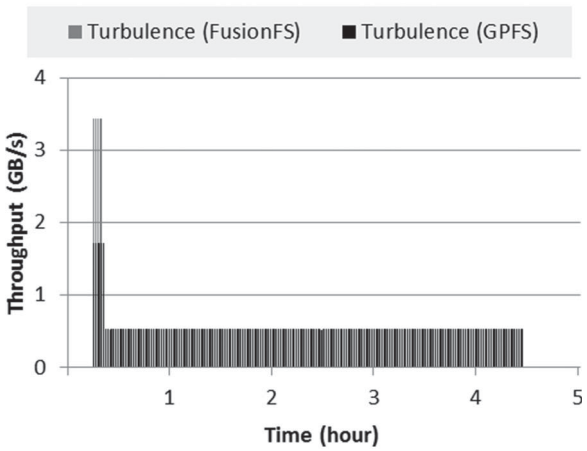


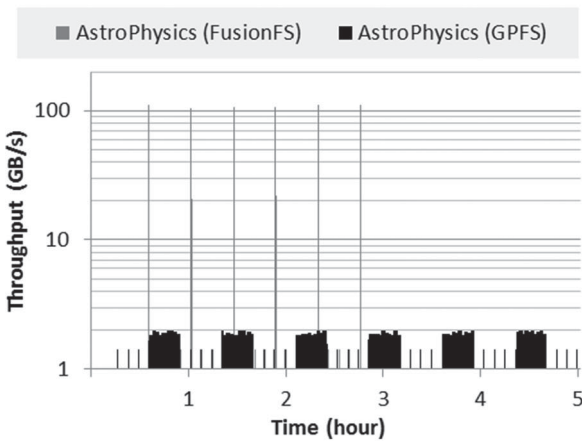
Fig. 11. Write throughput on Amazon EC2 Cloud.



(a) PlasmaPhysics



(b) Turbulence



(c) Astr oPhysics

Fig. 12. Top three write-intensive applications on Intrepid.

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's 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.

Fig. 12 shows the real-time I/O throughput of these workloads at 1,024-nodes. On FusionFS, these workloads

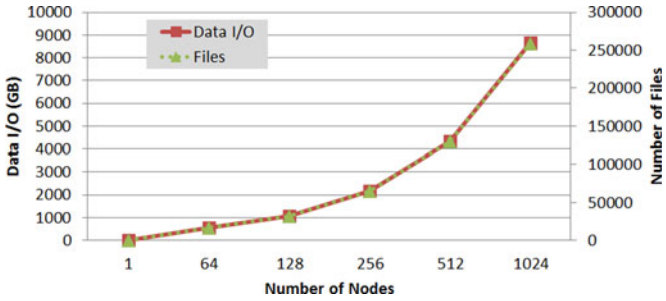


Fig. 13. The workload over three stages of BLAST.

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 2 GB/s while for FusionFS they reach over 100 GB/s. This increase in I/O performance accelerates the applications 2.1× times (PlasmaPhysics) and 1.6× 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, for example, ParallelBLAST [49]. In ParallelBLAST, the entire database (4 GB) 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 compare ParallelBLAST performance on FusionFS and GPFS with our AME (Any-scale MTC Engine) framework [50]. We carry out a weak scaling experiment of ParallelBLAST with 4 GB database on every 64-nodes, and increase 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 541 GB over 16,192 files for every 64-nodes. Fig. 13 shows the workload in all three stages and the number of accessed files from 1 node to 1,024 nodes. In our experiment of 1,024-node scale, the total I/O is about 9 TB applied to over 250,000 files.

As shown in Fig. 14, 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 32× speedup (at 512-nodes), and an average of 23× improvement between 64-nodes and 1,024-nodes. At 1-node scale, the GPFS kernel module is more effective in accessing an idle parallel file system. In FusionFS's case, the 1-node scale result involves the user-level FUSE module, which apparently causes BLAST to run 1.4× slower on FusionFS. However, beyond the corner-case of 1-node, FusionFS significantly outperforms GPFS. In particular, on 1,024-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.

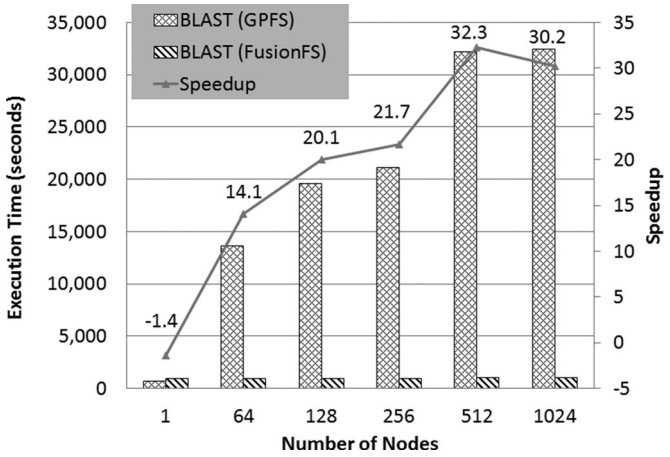


Fig. 14. BLAST execution time on Intrepid.

Based on the speedup of FusionFS and GPFS at different scales, we show the efficiency of both systems in Fig. 15. FusionFS still keeps a high efficiency (i.e., 64 percent) at 1,024-nodes scale, where GPFS falls below 5 percent at 64-nodes and beyond. For this application, GPFS is an extremely poor choice, as it cannot handle the concurrency generated by the application beyond 64-nodes. Recall that this GPFS file system has total 128 storage nodes in Intrepid, and is configured to support concurrent accessing from 40 K compute nodes, yet it exhibits little scaling from as small as 64-nodes.

Lastly, we measure the overall throughput of data I/O at different scales as generated by the BLAST application in Fig. 16. FusionFS has an excellent scalability reaching over 8 GB/s, and GPFS is saturated at 0.27 GB/s from 64 nodes and beyond.

4.5 Towards Exascales

While Section 4.3 reports FusionFS throughput and its comparison to other major file systems, this section concentrates on FusionFS's own performance and scalability at extreme scales. The first experiment is carried out on Intrepid on up to 16 K-nodes each of which has a FusionFS mount point. The workload is as follows: each client starts at the same time to write 16 GB data to FusionFS. Fig. 17 shows the real-time aggregate throughput of all 16 K nodes. FusionFS throughput shows about linear scalability: doubling the number of nodes

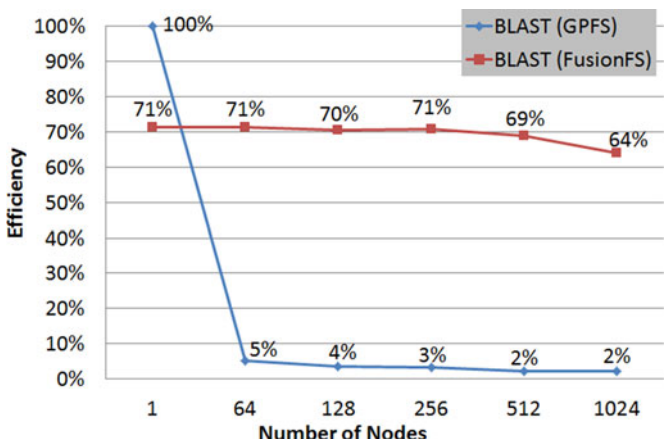


Fig. 15. BLAST I/O efficiency on Intrepid.

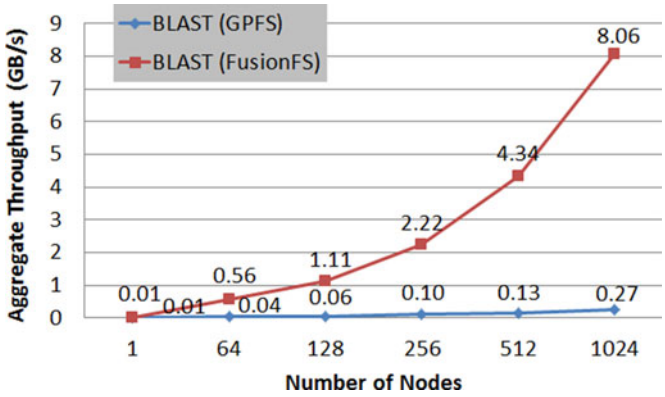


Fig. 16. BLAST I/O throughput on Intrepid.

yield doubled throughput. Specifically, we observe stable 2.5 TB/s throughput (peak 2.64 TB/s) on 16 K-nodes.

To show that FusionFS is scalable to even large scales (Intrepid has maximal 40 K nodes but requires a special reservation request to conduct experiments at such scales), we build a FusionFS simulator—FusionSim—based on the CODES framework [40]. In particular, we employ the torus network model, and simulate the data transfer between compute nodes as well as the local disk I/O. FusionSim simulates the same workload conducted in Fig. 17, and is validated by the real FusionFS trace on Intrepid on up to 16 K-nodes, as reported in Fig. 18. The error between the real trace and the simulation result is below 4 percent at all scales (512-nodes to 16 K-nodes), making FusionSim likely accurate in predicting the real system performance at larger scales. In fact, if we take into account the 5 percent variance from the experiments of FusionFS and FusionSim, the validation error is essentially negligible.

We scale FusionSim with the same workload in Fig. 17 to 2 million nodes, and report the throughput in Fig. 19. The ideal throughput is plotted based on the peak throughput 2.64 TB/s achieved at 16 K-node scale. FusionFS shows near linear scalability, with more than 95 percent efficiency at all scales. This can be best explained by its completely distributed metadata operations and the light network traffic involved in data I/O. In particular, FusionFS shows its

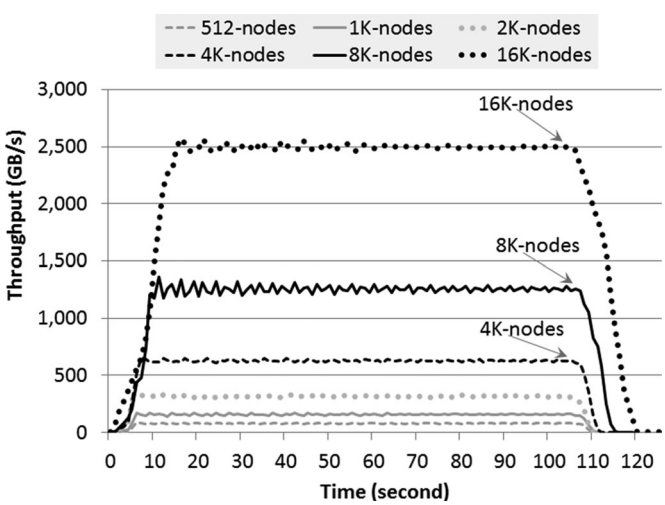


Fig. 17. FusionFS scalability on Intrepid.

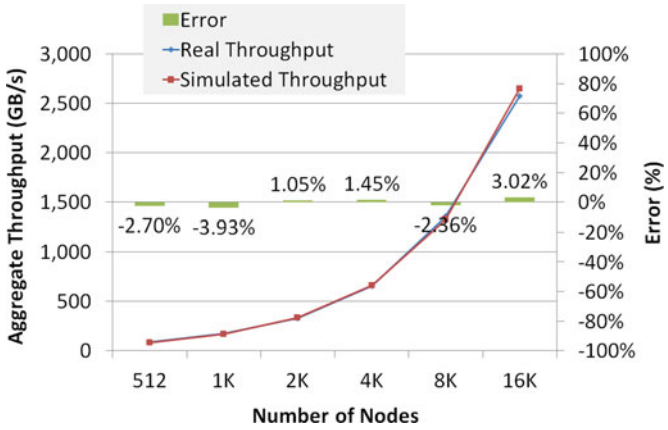


Fig. 18. Validation of FusionSim on Intrepid.

potential to achieve such an impressive I/O throughput of 329 TB/s on 2 million nodes.

In addition to the aggregate throughput, we feed FusionSim with more complex workloads regenerated from IOR benchmark [51] on GPFS. The workload is regenerated by the Darshan I/O tool [52] that is statistically equal to the real I/O load. This workload uses MPI collective I/O calls to a shared file in GPFS [19] on a leadership-class supercomputer Intrepid [22]. Each node (i.e., rank) does a sequence of 16 collective writes, closes the file, reopens, does a sequence of 16 collective reads, closes, then exits. All ranks open and close the shared file, and barrier before collective operations. Each rank moves 4 MB per call (64 MB total per rank), which gives us 1 TB total write, and 1 TB total read at 16 K scale. Note that this workload does not simply consist of independent I/Os, but involves a lot of collective communication such as data synchronization. We scale this workload from 16 K nodes to 1M nodes; so the maximal transferred data is 128 TB on 1 M nodes.

Results of IOR [51] workloads are shown in Fig. 20. For this particular workload, we observe that GPFS is inefficient because the aggregate throughput at the largest scale (i.e., 1 M) is only about 37 GB/s that is much smaller than the overall network bandwidth (88 GB/s) between GPFS and compute nodes. On the other hand, FusionFS is predicted to scale linearly and outperforms GPFS at all scales. Note that the speedup only slightly decreases from 16 K to 1 M nodes.

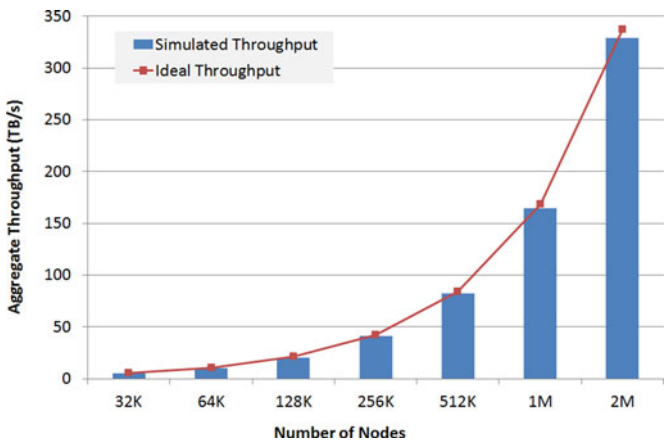


Fig. 19. Predicted FusionFS throughput by FusionSim.

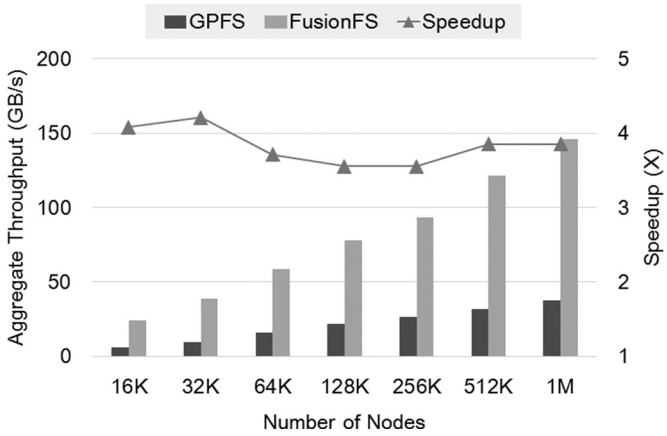


Fig. 20. Predicted FusionFS performance with IOR [51], comparing with GPFS.

5 Related Work

FusionFS's idea was first presented in [53] and then exposed in [54], [55], [56]. While the some results were recently published in [23], the system has been serving as the basis of related projects such as provenance tracking [57], [58], GPU-accelerated erasure coding [59], cooperative caching [26], [60], [61], and file compression [62], [63], [64].

There have been many shared and parallel file systems, such as the network file system (NFS [65]), general purpose file system (GPFS [19]), parallel virtual file system [20], and Lustre [66]. 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 [18], Hadoop File System [15], and Ceph [67]. However, many of these file systems are tightly coupled with execution frameworks (for example, MapReduce [68]), 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 operations at extreme scales. The majority of these systems do not expose the data locality information for general computational frameworks (for example, batch schedulers, workflow systems) to harness the data locality through data-aware scheduling.

The idea of distributed metadata can be traced back to xFS [69], but a central manager is in need to locate a particular file. Similarly, FDS [70] is proposed as a blob store on data centers, which 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, and demonstrates its scalability at extremes scales.

Co-location of compute and storage resources has attracted a lot of research interests. For instance, Nectar [71] automatically manages data and computation in data centers. From scheduler point of view, several work [72], [73] focused on exploring better data locality. 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.

6 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 new architecture brings, and build a distributed file system FusionFS to demonstrate how to address them. In particular, FusionFS is crafted to support extremely intensive metadata operations and is optimized for file writes. Extreme-scale evaluation on up to 16 K nodes demonstrates FusionFS's superiority over other popular storage systems for scientific applications. An event-driven simulator FusionSim, which is validated by FusionFS's traces, predicts that FusionFS will scale almost linearly towards the expected concurrency of the emerging exascale systems.

There are two major directions we plan to work on for the FusionFS file system. First, we will improve the load balance of FusionFS in an automatous manner by leveraging our prior work on incremental algorithms [74], [75], [76]. Second, we will support concurrent write in the next release of FusionFS.

Acknowledgments

This work was supported in part by the US National Science Foundation (NSF) under awards OCI-1054974 (CAREER). This work was also supported by the Department of Energy (DOE) Office of Advanced Scientific Computer Research (ASCR) under contract DE-AC02-06CH11357.

References

[1] T. Hey, S. Tansley, and K. Tolle, Eds., *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Redmond, CA, USA: Microsoft Res, 2009.

[2] P. A. Freeman, D. L. Crawford, S. Kim, and J. L. Munoz, "Cyberinfrastructure for science and engineering: Promises and challenges," *Proc. IEEE*, vol. 93, no. 3, pp. 682–691, Mar. 2005.

[3] D. Zhao, D. Zhang, K. Wang, and I. Raicu, "Exploring reliability of exascale systems through simulations," in *Proc. 21st ACM/SCS High Perform. Comput. Symp.*, 2013, p. 1.

[4] 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 *Proc. IEEE Symp. Mass Storage Syst. Technol.*, 2012, pp. 4–11.

[5] W. Tantisiriroj, 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 *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, p. 67.

[6] (2014). Apache Hadoop [Online]. Available: http://hadoop.apache.org/

[7] D. Boyuka, S. Lakshminarasimham, X. Zou, Z. Gong, J. Jenkins, E. Schendel, N. Podhorszki, Q. Liu, S. Klasky, and N. Samatova, "Transparent in situ data transformations in adios," in *Proc. 14th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2014, pp. 256–266.

[8] M. Gamell, I. Rodero, M. Parashar, J. C. Bennett, H. Kolla, J. Chen, P.-T. Bremer, A. G. Landge, A. Gyulassy, P. McCormick, S. Pakin, V. Pascucci, and S. Klasky, "Exploring power behaviors and trade-offs of in-situ data analytics," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, article 77.

[9] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2013, p. 78.

[10] F. Zhang, C. Docan, M. Parashar, S. Klasky, N. Podhorszki, and H. Abbasi, "Enabling in-situ execution of coupled scientific workflow on multi-core platform," in *Proc. IEEE Parallel Distrib. Process. Symp.*, 2012, pp. 1352–1363.

[11] J. C. Bennett, H. Abbasi, P.-T. Bremer, R. Grout, A. Gyulassy, T. Jin, S. Klasky, H. Kolla, M. Parashar, V. Pascucci, P. Pebay, D. Thompson, H. Yu, F. Zhang, and J. Chen, "Combining in-situ and in-transit processing to enable extreme-scale scientific analysis," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, article 49.

[12] (2014). DEEP-ER [Online]. Available: http://www.hpc.cineca.it/projects/deep-er

[13] 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 *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 155–163.

[14] (2014). Mira [Online]. Available: https://www.alcf.anl.gov/user-guides/mira-cetus-vesta

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. IEEE Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–10.

[16] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol.*, 2013, pp. 1–12.

[17] D. Nagle, D. Serenyi, and A. Matthews, "The Panasas activescale storage cluster: Delivering scalable high bandwidth storage," in *Proc. ACM/IEEE Conf. Supercomput.*, 2004, p. 53.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. ACM Symp. Oper. Syst. Principles*, 2003, pp. 29–43.

[19] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. 1st USENIX Conf. File Storage Technol.*, 2002, p. 19.

[20] P. H. Carns, W. B. Ligon, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000, pp. 317–327.

[21] P. Carns, S. Lang, R. Ross, M. Vilayannur, J. Kunkel, and T. Ludwig, "Small-file access in parallel file systems," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, May 2009, pp. 1–11.

[22] (2014). Intrepid [Online]. Available: https://www.alcf.anl.gov/user-guides/intrepid-challenger-surveyor

[23] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "FusionFS: Toward supporting data-intensive scientific applications on extreme-scale distributed systems," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 61–70.

[24] (2014). FUSE [Online]. Available: http://fuse.sourceforge.net

[25] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proc. ACM Symp. Appl. Comput.*, 2010, pp. 206–213.

[26] D. Zhao and I. Raicu, "HyCache: A user-level caching middleware for distributed file systems," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process. Workshops PhD Forum*, 2013, pp. 1997–2006.

[27] T. Li, X. Zhou, K. Wang, D. Zhao, I. Sadooghi, Z. Zhang, and I. Raicu, "A convergence of key-value storage systems from clouds to supercomputer," in *Proc. Concurr. Comput.: Pract. Exper.*, 2015.

[28] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu, "A dynamically scalable cloud data infrastructure for sensor networks," in *Proc. 6th Workshop Sci. Cloud Comput.*, 2015, pp. 25–28.

[29] T. Li, C. Ma, J. Li, X. Zhou, K. Wang, D. Zhao, and I. Raicu, "Graph/z: A key-value store based scalable graph processing system," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015.

[30] 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 *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2013, pp. 775–787.

[31] (2014). epoll [Online]. Available: http://man7.org/linux/man-pages/man7/epoll.7.html

[32] J. S. Plank, M. Blaum, and J. L. Hafner, "SD Codes: Erasure codes designed for how storage systems really fail," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 95–104.

[33] R. Rodrigues and B. Liskov, "High availability in DHTs: Erasure coding vs. replication," in *Proc. 4th Int. Conf. Peer-to-Peer Syst.*, 2005, pp. 226–239.

[34] Y. Gu and R. L. Grossman, "Supporting configurable congestion control in data transport services," in *Proc. ACM/IEEE Conf. Supercomput.*, 2005, p. 31.

[35] D. Zhao, K. Wang, K. Qiao, T. Li, I. Sadooghi, and I. Raicu, "Toward high performance storages through GPU encoding and locality-aware scheduling," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 1–12.

[36] 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 *Proc. IEEE Congr. Services*, 2007, pp. 199–206.
[37] 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 *Proc. ACM Int. Symp. High Perform. Distrib. Comput.*, 2009, pp. 207–216.
[38] C. D. Carothers, D. Bauer, and S. Pearce, "Ross: A high-performance, low memory, modular time warp system," in *Proc. 14th Workshop Parallel Distrib. Simul.*, 2000, pp. 53–60.
[39] N. Liu and C. D. Carothers, "Modeling billion-node torus networks using massively parallel discrete-event simulation," in *Proc. IEEE Workshop Principles Adv. Distrib. Simul.*, 2011, pp. 1–8.
[40] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothersy, and R. B. Ross, "CODES: Enabling co-design of multi-layer exascale storage architectures," in *Proc. Workshop Emerging Supercomput. Technol.*, 2011, pp. 303–312.
[41] N. Liu, C. Carothers, J. Cope, P. Carns, and R. Ross, "Model and simulation of exascale communication networks," *J. Simul.*, vol. 6, no. 4, pp. 227–236, 2012.
[42] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns, "Modeling a million-node dragonfly network using massively parallel discrete-event simulation," in *Proc. SC Companion: High Perform. Comput., Netw. Storage Anal.*, 2012, pp. 366–376.
[43] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, "Loggp: Incorporating long messages into the LOGP model—One step closer towards a realistic model for parallel computation," in *Proc. 7th Annu. ACM Symp. Parallel Algorithms Archit.*, 1995, pp. 95–105.
[44] Top500 (2014, Jun.). [Online]. Available: http://www.top500.org/list/2014/06/
[45] (2014). ZeptoOS [Online]. Available: http://www.mcs.anl.gov/zeptoos
[46] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde, "Falkon: A fast and light-weight task execution framework," in *Proc. ACM/IEEE Conf. Supercomput.*, 2007, pp. 1–12.
[47] (2014). Kodiak [Online]. Available: https://www.nmc-probe.org/wiki/Machines:Kodiak
[48] (2014). Amazon EC2 instance types [Online]. Available: http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html
[49] D. R. Mathog, "Parallel BLAST on split databases," *Bioinformatics*, vol. 19, no. 4, pp. 1865–1866, 2003.
[50] 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 *Proc. ACM/IEEE Conf. Supercomput.*, 2012, p. 85.
[51] (2014). IOR Benchmark [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks/IOR_summary_v1.0.pdf
[52] (2015). Darshan [Online]. Available: http://www.mcs.anl.gov/research/projects/darshan/
[53] D. Zhao and I. Raicu, "Distributed file systems for exascale computing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–2.
[54] D. Zhao, X. Yang, I. Sadooghi, G. Garzoglio, S. Timm, and I. Raicu, "High-performance storage support for scientific applications on the cloud," in *Proc. 6th Workshop Sci. Cloud Comput.*, 2015, pp. 33–36.
[55] D. Zhao and I. Raicu, "Storage support for data-intensive applications on extreme-scale HPC systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 1–2.
[56] D. Zhao and I. Raicu, "Storage support for data-intensive scientific applications on the cloud," in *Proc. NSF Workshop Experimental Support Cloud Comput.*, 2014, pp. 1–4.
[57] D. Zhao, C. Shou, T. Malik, and I. Raicu, "Distributed data provenance for large-scale data-intensive computing," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–8.
[58] C. Shou, D. Zhao, T. Malik, and I. Raicu, "Towards a provenance-aware distributed filesystem," in *Proc. 5th Workshop Theory Practice Provenance*, 2013, pp. 1–4.
[59] 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 *Proc. IEEE Int. Conf. Cluster Comput.*, 2013, pp. 1–5.
[60] D. Zhao, K. Qiao, and I. Raicu, "Towards cost-effective and high-performance caching middleware for distributed systems," *Int. J. Big Data Intell.*, 2015, pp. 1–5.
[61] D. Zhao, K. Qiao, and I. Raicu, "Hycache+: Towards scalable high-performance caching middleware for parallel file systems," in *Proc. IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput.*, 2014, pp. 1997–2006.
[62] D. Zhao, K. Qiao, J. Yin, and I. Raicu, "Dynamic virtual chunks: On supporting efficient accesses to compressed scientific data," *IEEE Trans. Services Comput.*, 2015.
[63] D. Zhao, J. Yin, K. Qiao, and I. Raicu, "Virtual chunks: On supporting random accesses to scientific data in compressible storage systems," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 231–240.
[64] D. Zhao, J. Yin, and I. Raicu, "Improving the I/O throughput for data-intensive scientific applications with efficient compression mechanisms," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal*, 2013, pp. 1–2.
[65] M. Eisler, R. Labiaga, and H. Stern, *Managing NFS and NIS*, 2nd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.
[66] P. Schwan, "Lustre: Building a file system for 1,000-node clusters," in *Proc. Linux Symp.*, 2003, pp. 1–7.
[67] 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 *Proc. 7th Symp. Oper. Syst. Design Implementation*, 2006, pp. 307–320.
[68] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. 6th USENIX Symp. Oper. Syst. Design Implementation*, 2004, pp. 137–150.
[69] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang, "Serverless network file systems," in *Proc. 15th ACM Symp. Oper. Syst. Principles*, 1995, pp. 109–126.
[70] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue, "Flat datacenter storage," in *Proc. USENIX Symp. Oper. Syst. Design Implementation*, 2012, pp. 1–15.
[71] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang, "Nectar: Automatic management of data and computation in datacenters," in *Proc. 9th USENIX Conf. Oper. Syst. Design Implementation*, 2010, pp. 75–88.
[72] Z. Zhou, X. Yang, D. Zhao, P. Rich, W. Tang, J. Wang, and Z. Lan, "I/O-aware batch scheduling for petascale computing systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015.
[73] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 119–128.
[74] D. Zhao and L. Yang, "Incremental isometric embedding of high-dimensional data using connected neighborhood graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 31, no. 1, pp. 86–98, Jan. 2009.
[75] R. Lohfert, J. Lu, and D. Zhao, "Solving SQL constraints by incremental translation to sat," in *Proc. 21st Int. Conf. Ind., Eng. Other Appl. Appl. Intell. Syst.*, 2008, pp. 669–676.
[76] D. Zhao and L. Yang, "Incremental construction of neighborhood graphs for nonlinear dimensionality reduction," in *Proc. Int. Conf. Pattern Recog.*, 2006, pp. 177–180.

Dongfang Zhao received the master's degree in computer science from Emory University, Atlanta. He is currently working toward the PhD degree in computer science at the Illinois Institute of Technology, Chicago. He has published more than 30 papers in the fields of big data, high-performance computing, distributed systems, cloud computing, and machine intelligence.

Ning Liu received the master's degree in control theory and application from Southeast University, China. He is currently working toward the PhD degree in computer science at the Illinois Institute of Technology focusing on parallel and distributed system. His current research interests include parallel computing, parallel simulation, storage systems, and MapReduce.

Dries Kimpe received the PhD degree in computer science from Katholieke Universiteit Leuven, Belgium. He is an assistant computer scientist in Argonne National Laboratory, an Institute fellow at Northwestern University, and an adjunct professor at the University of Illinois at Chicago. His research interests include high performance computing, fault tolerance, I/O forwarding, and MPI-IO.

Robert Ross received the PhD degree from Clemson University. He is a senior fellow in the Northwestern-Argonne Institute for Science and Engineering and in the University of Chicago/ Argonne Computation Institute, and an adjunct assistant professor in the Department of Electrical and Computer Engineering, Clemson University.

Xian-He Sun is a distinguished professor of computer science at the Illinois Institute of Technology and a guest faculty in the Mathematics and Computer Science Division, Argonne National Laboratory. His current research interests include parallel and distributed processing, memory and I/O systems, software systems for Big Data applications, and performance evaluation and optimization. He is a fellow of the IEEE.

Ioan Raicu received the PhD degree in computer science from the University of Chicago. He is an assistant professor of computer science at the Illinois Institute of Technology, as well as a guest research faculty in the Math and Computer Science Division, Argonne National Laboratory. His research interests include distributed systems, data-Intensive computing, cloud computing, and big data.