Taming Metadata Storms in Parallel Filesystems with MetaFS

Tim Shaffer
Department of Computer Science & Engineering
Notre Dame, Indiana
tshaffe1@nd.edu

Douglas Thain
Department of Computer Science & Engineering
Notre Dame, Indiana
dthain@nd.edu

ABSTRACT

Metadata performance remains a serious bottleneck in parallel filesystems. In particular, when complex applications start up on many nodes at once, a “metadata storm” occurs as each instance traverses the filesystem in order to search for executables, libraries, and other necessary runtime components. Not only does this delay the application in question, but it can render the entire system unusable by other clients. To address this problem, we present MetaFS, a user-level overlay filesystem that sits on top of an existing parallel filesystem. MetaFS indexes the static metadata content of complex applications and delivers it in bulk to execution nodes, where it can be cached and queried quickly, while relying on the existing parallel filesystem for data delivery. We demonstrate that MetaFS applied to a complex bioinformatics application converts the metadata load placed on a production Panasas filesystem from 1.1 million operations per task to 1.9 MB of bulk data per task, increasing the metadata scalability limit of the application from 66 nodes to 5,000 nodes.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**;

KEYWORDS

Parallel file systems, metadata scalability, parallel computing, scientific computing

ACM Reference format:
Tim Shaffer and Douglas Thain. 2017. Taming Metadata Storms in Parallel Filesystems with MetaFS. In *Proceedings of PDSW-DISCS'17: Second Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, Denver, CO, USA, November 12–17, 2017 (PDSW-DISCS'17),* 6 pages.
https://doi.org/10.1145/3149393.3149401

1 INTRODUCTION

As HPC facilities continue to increase computing capability by increasing core density and deploying hardware accelerators such as FPGAs, GPUs, and TPUs, the overall performance of an application depends more and more on the ability of the underlying storage system to keep up with the cluster. Parallel filesystems remain the tool of choice for managing storage, and are capable of delivering large file I/O bandwidth that scales with storage hardware by striping blocks, files, and volumes across different devices.

However, metadata performance remains a troublesome bottleneck for most storage systems. Access to file data can be accelerated through some combination of caching, striping, and larger transactions. But the same techniques are not so easily applied to metadata such as the directory structure and user-visible inode information, because data elements are small, require a high degree of consistency, and are manipulated using small update transactions.

Many production HPC applications generate periodic bursts of metadata access, particularly during application startup. While we often think of HPC applications as “simple” compiled executables that are loaded into cluster memory at startup, the reality is often more complicated. What the user thinks of as a single “application” may actually be a complex assembly of interpreted programs, dynamic libraries, configuration files, and calibration data that must be loaded via tens of thousands of interactions with the filesystem. If the same application is loaded simultaneously on thousands of nodes of the cluster, the result is a “metadata storm” as every node peppers the filesystems with thousands of small transactions. In this paper, we use a complex bioinformatics application (MAKER [5]) to show how this behavior can happen in practice.

This problem is difficult to solve in the general case, if we assume that the solution must observe the consistency semantics of general-purpose filesystems. However, we observe that a shared filesystem is used in different ways, each requiring somewhat different semantics: for example, data shared between concurrent processes requires strong, fine grained consistency; data shared between sequential processes requires strong, coarse grained consistency; and data that represents software will not change during the execution of a given task.

To solve the problem of metadata storms during application startup, we propose that the metadata representing the software be loaded in bulk to each node that requires it and cached for the duration of a single application run. Rather than modifying an existing parallel filesystem, we implemented this idea by creating an overlay filesystem (MetaFS) that can sit on top of an existing filesystem. MetaFS indexes all of the metadata for a particular application in a regular file, then transports the metadata in bulk to each execution node. Using a FUSE module at each node, metadata is served from the cache, while data is served from the original filesystem.

We performed an initial evaluation of this concept on a 24-node, 192-core cluster using a Panasas ActiveStor 16 filesystem with 77 nodes published to support up to 84 Gb/s read bandwidth and 94,000 IOPS while reading data. We performed an initial evaluation of MetaFS using a simple benchmark and observed a reduction in metadata operations from 179,091 to 8,738 I/O ops. When applied to the more complex application MAKER, metadata load was reduced from 1,142,781 to 14,726 I/O ops, which will enable the scalability of MAKER from 66 nodes to over 5,000 nodes.

When referring to "I/O operations", we include both metadata and data activity unless otherwise indicated. We also use the abbreviation IOPS for I/O ops. per second, and MIOPS and DIOPS for metadata and data activity, respectively.

2 BACKGROUND

Metadata behavior is critical to the performance of scientific applications at scale. Scientific software often uses shared filesystems to store intermediate files, synchronize between steps of an analysis, distribute application software, and collect results from multiple worker nodes. Each of these uses puts different types of strain on a general-purpose filesystem. Some use cases, such as distributing software components, leave room for tailored optimization.

Widely deployed shared filesystems such as Panasas [15], Lustre [2], Ceph [14], Gluster [1], and HDFS [11] use designated data and metadata servers to allow users to access programs and data from anywhere in the system. Servers for data tend to be simpler and optimized for throughput and parallel access. This allows fast access to bulk file data and is well suited to large reads and writes. Metadata servers provide hierarchical and consistent organization of files and directories. Before a node can read a file, metadata servers must resolve the file's path and determine where the file data resides. Efficient path resolution and metadata lookup is thus critical for the performance of a shared filesystem.

Generally, multiple metadata servers balance the load of requests by partitioning based on user activity or filesystem organization. If one or more of these nodes becomes strained, the remainder of the load must be shifted to other metadata nodes. When filesystem load becomes excessive, requests cannot be served efficiently and users see degraded performance or loss of service despite underutilized data storage nodes [16]. In general, metadata bottlenecks are the limiting factor for a parallel filesystem at scale. Modern parallel filesystems like Ceph are specifically designed to address pathological metadata access patterns.

Several approaches to this problem have been explored. One approach is to design a standalone metadata service distinct from the parallel filesystem which maps metadata storage tables to file objects in the parallel filesystem [10, 17]. This allows for the scaling up of the total metadata transaction rate of the system, but still requires each client to perform many transactions against the service. A complementary approach is to reduce the transaction rate between clients and servers by introducing new operations that access metadata in bulk or with weaker consistency guarantees. Example of this include the proposed `getlongdir` and `statlite` system calls [13], which are, unfortunately, not widely implemented.

Spindle [7] addresses the metadata problem in the specific case of loading shared object files. Spindle alters the behavior of the GNU dynamic loader to use an overlay network to distribute the load of data and metadata activity during program startup. While Spindle achieved significant performance improvement, it is not well suited for heterogeneous applications since each language or runtime (Python, Perl, Java, etc.) provides its own library loading facility.

As a case study to investigate this problem, we consider the MAKER bioinformatics pipeline. MAKER analyzes and creates annotated genomes from raw sequence data, and is widely used as the preliminary step of the analysis of population characterization and gene presence/activity. It can be deployed as a sequential, multicore, or MPI application, depending on the available resources. However, deploying MAKER at scale is a challenging task: it has a very large number of software dependencies that must be installed, such as OpenMPI, Perl 5, Python 2.7, RepeatMasker, BLAST, and several Perl modules. It also places unusual metadata loads on shared filesystems, and its scalability is limited in high latency environments.

From the user's perspective, MAKER is a single executable, but it consists of a large number of sub-programs written in different languages, each of which performs its own library loading and path searches. Figure 1 shows substantial bursts and spikes of metadata activity over the course of the analysis. Using `strace` logs, we measured the frequency of metadata-related I/O operations and found that MAKER exhibits extremely nonuniform filesystem behavior. The measured operations include library loads, program startup, and searches for reference data. Some of the largest spikes in metadata load occur in the first few seconds of the analysis. The bursts of activity near the middle and end of the analysis show more sustained activity patterns indicative of a large number of jobs starting together.

Assuming a "metadata storm" is a typical access pattern for a MAKER analysis, both the average metadata access rate and the peak activity are significant. As the number of nodes sending metadata requests with large spikes and bursts increases, the shared filesystem becomes more likely to see overlap of burst activity from different nodes. If the filesystem must handle frequent spikes in metadata activity from concurrent jobs, we expect the quality of service for all users to suffer. We experienced this problem first hand at Notre Dame when a single (well-meaning) user submitted a large batch of MAKER jobs to our campus HPC cluster and accidentally caused a metadata storm, rendering the entire facility unusable.

3 ANALYSIS

3.1 Metadata Behavior of MAKER

We suspected that these bursts of I/O activity were not caused by MAKER's outputs or intermediate data, so we examined execution logs of MAKER and found library search to be a major contributor to unruly behavior when the installation is located on a shared filesystem. Each time a MAKER command starts a new process, the library search makes numerous open calls before finding the required shared objects, Perl modules, etc. Framework initializations also result in frequent filesystem searches and similar metadata-intensive activity. Given the large number of processes spawned over the course of MAKER's analysis, these metadata operations can accumulate and put strain on the shared filesystem.

In order to examine MAKER's I/O behavior more carefully, we used `strace` to record the syscalls made during an analysis. We restricted our focus to I/O related syscalls, and divided them based on the areas of the filesystem they interacted with. The numerical

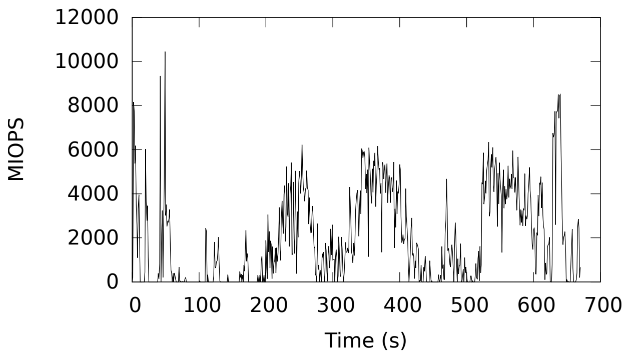

Figure 1: Metadata Intensity of MAKER.
The observed metadata rate over time to the shared filesystem from a single instance of MAKER analyzing E. coli on one node in 8 core MPI mode. For this analysis, the average metadata intensity was 1,975 MIOPS. The largest burst of activity, 10,454 MIOPS, occurs in the first minute of the analysis. Using MetaFS for this run would have reduced the average and peak metadata activity to the shared filesystem to 205 MIOPS and 1975 MIOPS, respectively.

	Access Mode	I/O Ops.	Bandwidth (bytes)
CWD	RW	257,060	1,435,228,808
TMP	RW	1,163,711	2,463,335,142
SW	RO	1,512,545	2,807,495,139
LOCAL	RO	906,327	68,929,672

Figure 2: I/O Activity by Filesystem Location.
This table gives I/O operations and read/write bandwidth observed during a MAKER analysis. CWD is the current working directory where the task's output data is stored, TMP is the local temporary disk for intermediate data, SW is the location of the installed software on a shared filesystem, and LOCAL indicates the libraries and executables provided by the local operating system installation.

results are shown in Figure 2. We observed that the majority of I/O operations were directed to the shared filesystem. Accesses to the local system and `/tmp` directory are not problematic here as they are local to each node and do not affect scalability. We decided to focus our attention on installation data on the shared filesystem, as we have can make stronger assumptions in this case. Specifically, the installation is shareable and read-only over the course of the workflow. Our optimizations also apply more generally in this case, as the behavior of installation files is largely independent of the algorithmic structure of the workflow.

3.2 Shared Filesystem Performance

The performance issues described previously occurred during a period of higher than normal system load. To study the performance under ideal conditions, we constructed a synthetic test to stress metadata I/O behavior. Our campus' administrators informed us that filesystem load was low to normal at the time of our tests. We created a directory tree containing 4,368 nested directories containing a total of 74,256 files. From a varying number of parallel hosts,

Parallel Nodes	Running Time (s)	Total Metadata IO Ops.	Average System MIOPS
1	13.7	179,091	13,038
4	22.6	716,364	31,664
8	41.9	1,432,728	31,194
16	86.1	2,865,456	33,262
24	130.6	4,298,184	32,916

Figure 3: Metadata I/O Operations under Ideal Conditions.
To measure the total metadata capability of the our filesystem, we ran the synthetic benchmark on an increasing number of hosts until saturation is reached. Each benchmark instance issues 179,091 metadata operations, and the whole system saturates at approximately 32K MIOPS.

we queried each file and directory (using `ls -lR`). The average time to traverse the directory tree is shown in Figure 3. Note that the kernel's filesystem cache was cleared before each measurement. This simulates dispatching a job to a worker node whose cache has not been warmed up. Kernel caching resolves I/O requests locally, but not on the initial access. We would prefer a solution that works for the first requests on new nodes instead of a simple caching approach. `strace` logs showed that a single traversal consists of 179,091 I/O-related syscalls that interact with the shared filesystem. Multiplying by the number of parallel hosts, we have the number of metadata I/O operations serviced by the shared filesystem over each parallel traversal. Dividing by the average time per traversal, we arrive at the metadata bandwidths shown in Figure 3. Beyond four concurrent tasks, the parallel filesystem appears to be saturated with metadata traffic at about 32K MIOPS.

We found that MAKER makes an average of 483 metadata I/O operations per second analyzing an *E. coli* dataset. To a first order approximation, the shared filesystem would be saturated by only 66 concurrent instance of MAKER, which is far less than needed for high-throughput bioinformatics research. This simple computation only considers the average metadata accesses. As shown in Figure 1, the distribution of I/O operations is far from uniform, and degraded performance could occur with even fewer instances if the bursts of activity coincide. Thus to scale up MAKER, we must reduce the metadata impact of the application.

4 POSSIBLE SOLUTIONS

There are several potential approaches to this problem using existing technologies. One would be to install the software stack natively on the local disk of each node in the system. This eliminates runtime scalability problems, but it also precludes most of the benefits of using a shared filesystem in the first place. Substantial storage is required at each node, and it becomes a burden on the system administrators to keep local disks synchronized. Further, different users of the system may have distinct software requirements that are mutually incompatible (e.g. Python 2 vs Python 3), so a single installation does not satisfy everyone.

Another possibility is to create a self-contained disk image with all the necessary files, store it in the shared filesystem, then mount it as needed on each node. This replaces a large number of files with a single image file. Once the image is mounted, both data and metadata accesses become data traffic to the shared filesystem. This approach would have the desired outcome of converting metadata traffic into cacheable data. However, it makes the overall system more difficult for the system administrator to manage and for the user to employ, because they must manually mount and unmount images in order to traverse the filesystem.

Container technologies such as Docker [9] and Singularity [8] are a more congenial way of generating and managing portable disk images, and also have the effect of efficiently transporting metadata along with data. However, Docker in particular requires system infrastructure and considerable local storage installed on each node, and does not exploit the performance and capacity of the existing parallel filesystem. Even assuming the system infrastructure supports Docker or another container technology, problems with metadata scalability can still arise. Unless each node stores a complete copy of the application, its dependencies, and any reference or input data, the applications themselves can create a metadata storm by loading reference data or similar files from the shared filesystem.

An ideal solution would retain the benefits of using a shared filesystem and keep the burden of manually transferring data or building large, immutable disk images away from researchers. Thus we looked for an approach that allows researchers to use a shared filesystem normally, but can take advantage of some inside knowledge of the application to improve metadata performance. An overlay filesystem is an effective technique for changing the performance behavior of applications without making intrusive changes to the applications or infrastructure. For example, PLFS [3] is an overlay which maps a large checkpoint file into multiple independent files, resulting in a large performance improvement without changing the underlying filesystem. In this case, we aim to design an overlay filesystem which converts a large number of metadata operations into a smaller number of data operations which are more easily cached and distributed. Our goal is not to solve *all* metadata-related performance issues. Instead, we chose to focus on a particular case that occurs frequently in scientific workflows and is likely to be the first performance barrier researchers hit when scaling up computational analyses. Our case study with MAKER illustrates how the overhead of simply loading libraries and reference data becomes problematic with a large number of nodes working in parallel. Using a metadata index, we reduce load on the shared filesystem and free up time for researchers to focus on the algorithmic activity in their workflows.

5 METAFS

MetaFS is an overlay filesystem that accelerates read-only metadata activity in a parallel filesystem. Application software does not need to be modified or configured with different paths. A simple indexing script reads all metadata information from MAKER's installation directory and builds an index file to be used throughout the workflow. MetaFS allows reads of file data to pass through unmodified, as shown in Figure 4. For metadata operations like `stat` and directory listings, however, results come from the locally cached index of the installation. MetaFS can handle metadata queries and negative lookups without interacting with the shared filesystem at all.

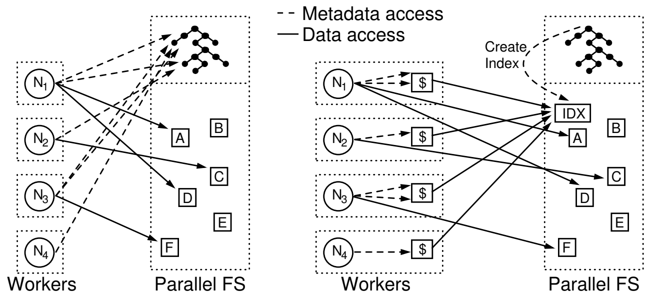

Figure 4: Architecture of MetaFS.
N_1, N_2, N_3, N_4 refer to processes on the worker nodes. The parallel filesystem holds a metadata tree (shown at the top) and file data (shown below). On the left, processes directly access the shared filesystem, which is the usual system configuration. On the right, MetaFS uses a cached copy of the metadata index (marked IDX) to service metadata requests locally. MetaFS reads the index data from the shared filesystem at startup. The metadata index is created once for the whole system before starting MetaFS on workers.

MetaFS is derived from an earlier system, GROW-FS [6], which was designed as a lightweight, read-only filesystem for wide-area distribution. MetaFS uses approaches to metadata management and software distribution pioneered by GROW-FS and CVMFS [4], but applies them to parallel filesystems at the scale of a shared computing site. Using FUSE, any access to the MAKER installation is transparently redirected to MetaFS. No modifications to MAKER are necessary.

FUSE is widely available, but on many shared computing resources, mounting via FUSE is only available for privileged users. We implemented MetaFS using FUSE because it provides a lightweight, low-overhead interface for a virtual filesystem. We considered overriding parts of libc to redirect metadata-related I/O, but in our experience this approach is very brittle and coupled to a specific application. We also considered implementing MetaFS using Parrot [12], which uses the Linux kernel's `ptrace` interface to intercept syscalls and is purely user-level, but doing so introduces a larger performance overhead.

MetaFS's metadata index transparent from the application's perspective. This transparency was important in MAKER as several of the programs use hard coded paths to the installation, which precludes moving the installed directories. By mounting MetaFS on top of the shared filesystem directory it is serving, the system appears unchanged from the application's perspective. The processes on the system interact with the filesystem as usual, but the kernel redirects these open, read, and other calls to the FUSE module running in user space, which is free to handle calls as it sees fit. The MetaFS module mediates access to the shared filesystem so that misbehaving applications cannot directly interact with the shared filesystem and overload the system or impact other users.

6 EVALUATION

Figure 5 shows the reduction in metadata operations on the shared filesystem for MAKER and for the synthetic benchmark. With MetaFS in place, we observed (as expected) that the majority of

	Metadata Ops.	Data Transfer (B)
`ls`	179,091	0
`ls` + MetaFS	8,738	4,900,655
MAKER	1,142,781	2,807,495,139
MAKER + MetaFS	14,726	2,809,472,114

Figure 5: Reduction in Metadata Load on the Shared Filesystem with MetaFS.
Without MetaFS, the synthetic benchmark performs a large number of metadata operations and data operations. With MetaFS, the majority of metadata operations are replaced by a single read of the index file. MAKER shows a similar trade of metadata operations for data transfer. In MAKER's case, data operations in the analysis dwarf the transfer of the index file.

metadata requests are served locally, while data transfer increases due to loading the index file. Note that this measurement was taken with a cold filesystem cache. In the case of a warm cache, we observed that MetaFS's performance is on par with direct access to the shared filesystem. FUSE introduces another layer of indirection when accessing the filesystem, so we were concerned that it would significantly worsen performance with a warm cache. We did not observe a significant performance decrease on either the synthetic benchmark or real MAKER analyses.

The synthetic benchmark illustrates some important considerations for running MAKER at scale. First, local caching is important for metadata-intensive operations in parallel. The cold cache measurements would apply when submitting jobs to previously unused worker nodes. In this case a large number of new workers starting together could bring the system to a crawl, with performance across the system suffering and no obvious culprit. The warm cache case can also lead to surprising performance degradation when scaling up. If a researcher had been testing on a small set of workers, the local caches could hide the impact of metadata activity. On scaling out to new machines that have data from other users in cache, the performance could sharply decrease despite efficient algorithmic implementation.

MAKER functioned as usual with MetaFS in place over its installation files. No modifications to MAKER were necessary. Comparing the performance of a single instance of MAKER running with and without MetaFS, we did not observe appreciable overhead. Despite the additional indirection on filesystem access which would be visible on microbenchmarks, the overall running time of a real analytic pipeline appears not to be affected. This is readily explained by an efficient scheduler on the system that can hide small delays in I/O by switching to another process in the workflow. Based on `strace` logs and statistics collected by MetaFS during MAKER's analysis, we observe a significant reduction in metadata operations that reach the parallel filesystem as shown in Figure 5. With MetaFS handling metadata requests locally, MAKER makes an average of 6 metadata-related I/O operations per second. This is substantial improvement over the previously observed 483 MIOPS. All directory listings, file stats, etc. are transparently handled locally without interacting with the shared filesystem. MetaFS can also locally service the frequent failed opens during library searches. The shared filesystem does not need to receive opens for files MetaFS knows not to exist, so MetaFS can absorb all metadata requests aside from successful opens and closes. Thus with MetaFS in place, the shared filesystem could support up to 5,000 parallel instances of MAKER under ideal conditions. We do not claim that MetaFS alone allows 5,000 instances to run in parallel, but only to have removed one of the barriers to scalability. Applications are likely to face other scalability limits, but the metadata traffic from program loading that MetaFS targets is common across applications. It is also a fairly low limit that researchers are likely to meet while scaling up an analysis.

The primary difficulties with MetaFS were index creation and read-only access. Before using MetaFS, users must generate an index for the mount directory. The index for the synthetic directory tree took 323 seconds to generate and occupied 4.7 MB disk space. For comparison, the index of MAKER's installation directory took 129 seconds to generate and 1.9 MB disk space. Assuming that the installation does not change over the course of a workflow, the index only needs to be generated once. When starting MetaFS, the index file is read in its entirety by each node. Nodes can start up immediately with the same metadata index until the user changes or updates the software installation. Since parallel filesystems are well suited to bulk parallel reads, we are happy to replace numerous metadata requests with reads of data. As shown in Figure 5, this additional read is insignificant compared to the data transferred over the course of the analysis.

After any changes to the directory, the index must be updated. For directories that change infrequently, this is no problem. Ensuring read-only access is also an important consideration for users. The current version of MetaFS sidesteps issues of consistency by blocking writes to the indexed directory. Thus users must be sure that their workflows *only* read program or reference data. For this work we performed detailed syscall-level analysis of MAKER's behavior. Other researchers would likewise need detailed knowledge of the I/O behavior of their software stacks.

7 CONCLUSION

This work does not attempt to address the general problem of handling metadata access in a parallel filesystem, instead targeting the specific case of bursts of metadata activity during program loading. By caching a metadata index on each worker node, we traded metadata activity for data transfer and observed order of magnitude decreases in metadata load on the shared filesystem. We plan to test our approach on other metadata-intensive software, and to verify the scalability of our approach using real applications like MAKER at larger scale. The decision to use FUSE makes it more complicated to operate at scale on shared computing resources, so adding support for other implementations mentioned would ease deployment. The greatest difficulty from a researcher's perspective is determining where to apply optimizations in a complex scientific workflow. To this end, we would also like to automate the profiling we performed on MAKER to identify problematic patterns of activity, allowing researchers to more readily understand the I/O behavior of their workflows.

REFERENCES

[1] 2017. Gluster. (2017). http://www.gluster.org/

[2] R. Behrends, L. K. Dillon, S. D. Fleming, and R. E. K. Stirewalt. 2007. *White paper: LUSTRE FILE SYSTEM High-Performance Storage Architecture and Scalable Cluster File System.* Technical Report. Sun Microsystems, Menlo Park, California. 20 pages.

[3] J. Bent, G. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate. 2009. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. 1–12. https://doi.org/10.1145/1654059.1654081

[4] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann. 2011. CernVM-FS: Delivering Scientific Software to Globally Distributed Computing Resources. In *Proceedings of the First International Workshop on Network-aware Data Management (NDM '11)*. ACM, New York, NY, USA, 49–56. https://doi.org/10.1145/2110217.2110225

[5] M. S. Campbell, C. Holt, B. Moore, and M. Yandell. 2014. Genome Annotation and Curation Using MAKER and MAKER-P. *Curr Protoc Bioinformatics* 48 (Dec 2014), 1–39.

[6] Gabrielle Compostella, Simone Pagan Griso, Donatella Lucchesi, Igor Sfiligoi, and Douglas Thain. 2009. CDF Software Distribution on the Grid using Parrot. In *Computing in High Energy Physics*.

[7] Wolfgang Frings, Dong H. Ahn, Matthew LeGendre, Todd Gamblin, Bronis R. de Supinski, and Felix Wolf. 2013. Massively Parallel Loading. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 389–398. https://doi.org/10.1145/2464996.2465020

[8] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE* 12, 5 (05 2017), 1–20. https://doi.org/10.1371/journal.pone.0177459

[9] Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014). http://dl.acm.org/citation.cfm?id=2600239.2600241

[10] K. Ren, Q. Zheng, S. Patil, and G. Gibson. 2014. IndexFS: Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*. 237–248. https://doi.org/10.1109/SC.2014.25

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST) (MSST '10)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/MSST.2010.5496972

[12] Douglas Thain and Miron Livny. 2003. Parrot: Transparent User-Level Middleware for Data Intensive Computing. In *Workshop on Adaptive Grid Middleware at PACT*.

[13] Murali Vilayannur, Samuel Lang, Robert Ross, Ruth Klundt, Lee Ward, et al. 2008. Extending the POSIX I/O interface: A parallel file system perspective. *Argonne National Laboratory, Tech. Rep. ANL/MCS-TM-302* (2008).

[14] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. 2006. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 307–320.

[15] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. 2008. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, Berkeley, CA, USA, Article 2, 17 pages. http://dl.acm.org/citation.cfm?id=1364813.1364815

[16] Bing Xie, Jeffrey Chase, David Dillow, Oleg Drokin, Scott Klasky, Sarp Oral, and Norbert Podhorszki. 2012. Characterizing Output Bottlenecks in a Supercomputer. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 8, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389007

[17] Q. Zheng, K. Ren, and G. Gibson. 2014. BatchFS: Scaling the File System Control Plane with Client-Funded Metadata Servers. In *2014 9th Parallel Data Storage Workshop*. 1–6. https://doi.org/10.1109/PDSW.2014.7