ADA-FS—Advanced Data Placement via Ad hoc File Systems at Extreme Scales

Sebastian Oeste, Marc-André Vef, Mehmet Soysal, Wolfgang E. Nagel, André Brinkmann, and Achim Streit

Abstract Today's High-Performance Computing (HPC) environments increasingly have to manage relatively new access patterns (e.g., large numbers of metadata operations) which general-purpose parallel file systems (PFS) were not optimized for. Burst-buffer file systems aim to solve that challenge by spanning an ad hoc file system across node-local flash storage at compute nodes to relieve the PFS from such access patterns. However, existing burst-buffer file systems still support many of the traditional file system features, which are often not required in HPC applications, at the cost of file system performance.

The ADA-FS project aims to solve that challenge by providing a temporary burst-buffer file system—GekkoFS—which relaxes POSIX, based on previous usage studies of how HPC applications use file systems. Due to a highly distributed and decentralized design GekkoFS reaches scalable data and metadata performance with tens of millions of metadata operations per second on a 512 node cluster. The ADA-FS project further investigated the benefits of using ad hoc file systems and how they can be integrated into the workflow of supercomputing environments. In addition, we explored how to gather application-specific information to optimize the file system for an individual application.

S. Oeste (✉) · W. E. Nagel
TU Dresden, Dresden, Germany
e-mail: sebastian.oeste@tu-dresden.de; wolfgang.nagel@tu-dresden.de

M.-A. Vef · A. Brinkmann
Johannes Gutenberg University Mainz, Mainz, Germany
e-mail: vef@uni-mainz.de; brinkman@uni-mainz.de

M. Soysal · A. Streit
KIT Karlsruhe, Karlsruhe, Germany
e-mail: mehmet.soysal@kit.edu; achim.streit@kit.edu

1 Introduction

Application-imposed workloads on *High-Performance computing* (HPC) environments have considerably changed in the past decade. While traditional HPC applications have been compute-bound, large-scale simulations, today's HPC applications are also generating, processing, and analyzing massive amounts of experimental data—known as *data-driven science* applications—affecting several scientific fields. Some of which have already made significant progress in previously unaddressable challenges due to newly discovered techniques [27, 55].

Many data-driven workloads are based on new algorithms and data structures which impose new requirements on HPC file systems [45, 77]. Particularly, large numbers of metadata operations, data synchronization, non-contiguous and random access patterns, and small I/O requests [14, 45], used in data-driven science applications, are challenging for today's general-parallel file systems (PFSs) to handle since past workloads mostly perform sequential I/O operations on large files. Not only are such applications disruptive to the shared storage system but also heavily interfere with other applications which access the same shared storage system [18, 68]. As a result, many workloads which impose these new types of I/O operations suffer from prolonged I/O latencies, reduced file system performance, and occasional long wait times.

Software-based approaches, e.g., application modifications or middleware and high-level libraries [21, 39], and hardware-based approaches, moving from magnetic disks to NAND-based solid-state drives (SSDs) within PFSs, are attempts to mitigate the impact of these new access patterns on the HPC system. However, software-based approaches often suffer from time-consuming adaptations within applications and are sometimes (based on the underlying algorithms) even impossible to adapt to. One of the hardware-based approaches leverages on, nowadays, existing SSDs, installed within a compute node, in order to use them as *node-local* burst buffers. To achieve high metadata performance, they can be deployed in combination with a dynamic *burst buffer file system* [5, 78]. Nonetheless, existing burst buffer file systems have been mostly POSIX compliant which can severely reduce a file system's peak performance [75].

The ADA-FS project, funded by the German Research Foundation (DFG) through the Priority Programme 1648 “Software for Exascale Computing”, aims to further explore the possibilities of burst buffer file systems in this context while investigating how they can be used in a modern HPC system. The developed burst buffer file system—*GekkoFS*—acts as ADA-FS' main component. *GekkoFS* is a temporarily deployed, highly-scalable distributed file system for HPC applications which aims to accelerate I/O operations of common HPC workloads that are challenging for modern PFSs. As such, it can be used in several temporary use cases, such as the lifetime of a compute job or in longer-term use cases, e.g., campaigns. Unlike previous works on burst buffer file systems, it relaxes POSIX by removing some of the semantics that most impair I/O performance in a distributed context and takes previous studies on the behavior of HPC applications into account [37] to

optimize the most used file system operations. As a result, GekkoFS reaches scalable data and metadata performance with tens of millions of metadata operations per second on a 512 node cluster while still providing strong consistency for file system operations that target a specific file or directory. In fact, due to its highly distributed and decentralized file system design, GekkoFS is built to perform on even bigger supercomputers, as exascale environments are right around the corner.

While GekkoFS provides the core building block within ADA-FS, it relies and benefits from further information of the application it is used with. Application-specific information that we gather can then further optimize the file system (e.g., the used file system block size) and therefore may increase the file system's performance in terms of latency and throughput. In addition, the ADA-FS project investigated how such a temporary and *on demand* burst buffer file system can be integrated into the workflow of batch systems in supercomputing environments. Although it is hard to reliably predict when compute jobs finish to prematurely deploy GekkoFS for a following ADA-FS job, for instance, we investigated and showed the benefits of on demand burst buffer file systems concerning both application performance and the reduction of the PFS load as a result of using such a file system.

The article is structured as follows: first, we describe GekkoFS' design and its evaluation of nowadays common and challenging HPC workloads on a 512 node cluster in Sect. 2. Section 3 discusses the existing challenges when data is staged in advance and how we solved the challenge, through implementing a plugin for the batch system. Section 4 discusses how we can detect system resources like the amount of node local storage or the NUMA configuration of a node which can be used for the deployment of the GekkoFS file system even on heterogenous compute nodes. In Sect. 5 we show how the option for an on demand file system, can be added to an HPC system. We follow with an evaluation of the performance of GekkoFS for new NVME based storage systems in Sect. 6. Finally, we conclude in Sect. 7.

2 GekkoFS—A Temporary Burst Buffer File System for HPC

In this section, we present the main component of ADA-FS—GekkoFS. GekkoFS is a temporarily deployed, highly-scalable burst buffer file system for HPC applications. In general, the goal of GekkoFS is to accelerate I/O operations in common HPC workloads that are challenging for modern PFSs while offering the combined storage capabilities of node-local storage devices. Further, it does not only aim for providing scalable I/O performance, but, in particular, focuses on offering scalable metadata performance by departing from traditional ways of handling metadata in distributed file systems. To provide a single, global namespace, accessible to all file system nodes, the file system pools together fast node-local storage resources of all participating file system nodes.

Based on previous studies [37] on the behavior of HPC applications, GekkoFS relaxes or removes some of the POSIX semantics, known to heavily impact I/O performance in a distributed environment. As a result, it is able to optimize for the most used file system operations, achieving tens of millions of metadata operations per second on a 512 node cluster. At the same time, GekkoFS is able to run complex applications, such as OpenFOAM solvers [32], and since the file system runs in user-space and it can be easily deployed in under 20 s on a 512 node cluster, it is usable by any user. Consequently, GekkoFS can be used for several use cases which require an ephemeral distributed file system, such as during the lifetime of a compute job or campaigns where data is simultaneously accessed by many nodes in short bursts.

Parts of this section’s contents is build on the conference paper by the authors M.-A. Vef et al. [72] and the journal article by the authors M.-A. Vef et al. [71] which both discuss each of the system components of GekkoFS in more detail and provide an in-depth investigation into the performance of GekkoFS compared to other file systems in various HPC environments. First, Sect. 2.1 provides a background on parallel and distributed file systems and discusses some of the related work in the context of burst buffer file systems. Section 2.2 presents the file system’s core architecture and design to achieve scalable data and metadata performance in a distributed environment. Finally, in Sect. 2.3 we demonstrate GekkoFS data and metadata performances.

2.1 *Related Work*

In this section, we give an overview over existing HPC file systems and discuss the differences to GekkoFS.

2.1.1 **General-Purpose Parallel File Systems**

Most HPC systems are equipped with a backend storage system which is globally accessible using a parallel file system (e.g., GPFS [57], Lustre [7, 53], BeeGFS [26], or PVFS [56]). These file systems offer a POSIX-like interface and focus on data consistency and long-term storage. However, due to the nature of the file system being globally accessible, single applications can disrupt the I/O performance of other applications as well. In addition, these file systems are not well suited for small file accesses, in particular on shared files, often found in scientific applications [45].

The design of GekkoFS does not focus on long-term storage and aims for temporary use cases, such as in the context of compute jobs or campaigns. In addition, since GekkoFS relaxes POSIX semantics, it is able to provide a significant increase in metadata performance.

2.1.2 Node-Local Burst Buffers

Burst buffers are fast, intermediate storage systems that aim to reduce the load on the global file system and on reducing an applications' I/O overhead [38]. Such burst buffers can be categorized into two groups [78]: remote-shared and node-local. Remote-shared burst buffers are generally dedicated I/O nodes to forward application I/O to the underlying PFS, e.g., DDN's IME¹ and Cray's DataWarp.²

Node-local burst buffers, on the other hand, are collocated with compute nodes, using existing node-local storage. This node-local storage is then used to create a (distributed) file system which spans over a number of nodes for the lifetime of a compute job, for example. Node-local burst buffers can also be dependent on the PFS (e.g., PLFS [5]) or are sometimes even managed directly by the PFS [49].

BurstFS [78], perhaps the most related work to ours, is a standalone burst buffer file system which does not require a centralized instance as well. However, GekkoFS is not limited to writing data locally like BurstFS. Instead, all data is distributed across all participating file system nodes to balance data workloads for write and read operations without sacrificing scalability. BeeOND [26] can create a job-temporal file system on a number of nodes similar to GekkoFS. BeeOND is, in contrast to our file system, POSIX compliant and our GekkoFS measurements show a much higher metadata throughput than offered by BeeOND [69, 71].

2.1.3 Metadata Scalability

The management of inodes (containing a file's metadata) and related directory blocks (containing data about which files belong to the directory) are the main scalability limitations of file systems in a distributed environment [73]. Typically, general-purpose PFSs distribute data across all available storage targets. While this technique works well for data, it does not achieve the same throughput when handling metadata [11, 54], although the file system community presented various techniques to tackle this challenge [5, 22, 50, 51, 79, 80]. The performance limitation can be attributed to the sequentialization enforced by underlying POSIX semantics which is particularly degrading throughput when an extremely large number of files is created in a single directory from multiple processes. This workload, common to HPC environments [5, 49, 50, 74], can become an even bigger challenge for upcoming data-science applications. GekkoFS handles directories and replaces directory entries by objects, stored within a strongly consistent key-value store which helps to achieve tens of millions of metadata operations for billions of files.

¹IME: <https://www.ddn.com/products/ime-flash-native-data-cache/>.

²Datawarp: <https://www.cray.com/datawarp>.

2.2 Design

In this section, we present goals, architecture, and general design of GekkoFS which allows scalable data and metadata performance. In general, any user without administrative access should be able to deploy GekkoFS. The user dictates on how many compute nodes and at which path the mountpoint of GekkoFS and its metadata and data is stored. The user is then presented with a single global namespace, consisting of the aggregated node-local storage of each node. To provide this functionality GekkoFS aims to achieve four core goals:

Scalability: GekkoFS should be able to scale with an arbitrary number of nodes and efficiently use available hardware.

Consistency model: GekkoFS should provide the same strong consistency as POSIX for common file system operations that access a specific data file. However, the consistency of directory operations, for example, can be relaxed.

Fast deployment: To avoid wasting valuable and expensive resources in HPC environments, the file system should startup within a minute and be ready for usage immediately by applications after the startup succeeds.

Hardware independence: GekkoFS should be able to support networking hardware that is commonly used in HPC environments, e.g., Omni-Path or Infiniband. The file system should be able to use the native networking protocols to efficiently move data between file system nodes. Finally, GekkoFS should work with modern and future storage technologies that are accessible to a user at an existing file system path.

2.2.1 POSIX Semantics

Similarly to PVFS [12] and OrangeFS [42], GekkoFS does not provide complex global locking mechanisms. In this sense, applications should be responsible to ensure that no conflicts occur, in particular, concerning overlapping file regions. However, the lack of distributed locking has consequences for operations where the number of affected file system objects is unknown beforehand, e.g., `readdir()` called by the `ls -l` command. In these *indirect file system operations*, GekkoFS does not guarantee to return the current state of the directory and follows the eventual-consistency model. Furthermore, each file system operation is synchronous without any form of caching to reduce file system complexity and to allow for an evaluation of its raw performance capabilities.

Further, GekkoFS does not support move or rename operations or linking functionality as HPC application studies have shown that these features are rarely or not used at all during the execution of a parallel job [37]. Such unsupported file system operations then trigger an I/O error to notify an application. Finally, security management in the form of access permissions is not maintained by GekkoFS since it already implicitly follows the security protocols of the node-local file system.

Fig. 1 GekkoFS architecture

2.2.2 Architecture

The architecture of GekkoFS (see Fig. 1) consists of two main components: a client library and a server process. An application that uses GekkoFS must first preload the client interposition library which intercepts all file system operations and forwards them to a server (*GekkoFS daemon*), if necessary. The GekkoFS daemon, which runs on each file system node, receives forwarded file system operations from clients and processes them independently, sending a response when finished. In the following paragraphs, we describe the client and daemon in more detail.

2.2.3 GekkoFS Client

The client consists of three components: (1) An interception interface that catches relevant calls to GekkoFS and forwards unrelated calls to the node-local file system; (2) a file map that manages the file descriptors of open files and directories, independently of the kernel; and (3) an RPC-based communication layer that forwards file system requests to local/remote GekkoFS daemons.

Each file system operation is forwarded via an RPC message to a specific daemon (determined by hashing of the file's path, similar to Lustre DNE ³) where it is directly executed. In other words, GekkoFS uses a pseudo-random distribution to spread data and metadata across all nodes, also known as *wide-striping*. Because each client is able to independently resolve the responsible node for a file system operation, GekkoFS does not require central data structures that keep track of where metadata or data is located. To achieve a balanced data distribution for large files,

³<https://lustre.ornl.gov/ecosystem-2016/documents/papers/LustreEco2016-Simmons-DNE.pdf>.

data requests are split into equally sized chunks before they are distributed across file system nodes (or GekkoFS daemons). The GekkoFS daemons then store each received chunk in a separate file (so-called *chunk files*) in its underlying node-local storage. If supported by the underlying network fabric protocol, the client exposes the relevant chunk memory region to the daemon, accessed via *remote-direct-memory-access* (RDMA).

2.2.4 GekkoFS Daemon

GekkoFS daemons consist of three parts: (1) A key-value store (KV store) used for storing metadata; (2) an I/O persistence layer that reads/writes data from/to the underlying local storage system; and (3) an RPC-based communication layer that accepts local and remote connections to handle file system operations.

Each daemon operates a single local RocksDB KV store [17]. RocksDB is optimized for NAND storage technologies with low latencies and fits GekkoFS' needs as SSDs are primarily used as node-local storage in today's HPC clusters. While RocksDB fits this use case well, the component is replaceable by other software or hardware solutions. Therefore, GekkoFS may introduce various choices for backends in the future to, for example, support recent key-value SSDs⁴

For the communication layer, we leverage on the *Mercury* RPC framework [62]. It allows GekkoFS to be network-independent and to efficiently transfer large data within the file system. Within GekkoFS, Mercury is interfaced indirectly through the *Margo* library which provides *Argobots*-aware wrappers to Mercury's API with the goal to provide a simple multi-threaded execution model [13, 58]. Using Margo allows GekkoFS daemons to minimize resource consumption of Margo's progress threads and handlers which accept and handle RPC requests [13].

Further, as indicated in Sect. 2.1.3, GekkoFS does not use a global locking manager. Therefore, when multiple processes write to the same file region concurrently, they may cause a shared write conflict with resulting undefined behavior with regards to which data is written to the underlying node-local storage. Such conflicts can, however, be handled locally by any GekkoFS daemon because it is using a POSIX-compliant node-local file system to store the corresponding data chunks, serializing access to the same chunk file. Note that such conflicts in a single file only affect one chunk at a time since the file's data is spread across many chunk files in the file system. As a result, chunks of that file are not disrupted during such a potential shared write conflict.

⁴https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

2.3 Evaluation

In this section, we evaluate the performance of GekkoFS based on various unmodified microbenchmarks which catch access patterns that are common in HPC applications. First, we describe the experimental setup and introduce the workloads that we simulate with microbenchmark applications. Then, we investigate the startup time of GekkoFS and compare metadata performance against a Lustre parallel file system. Although GekkoFS and Lustre have different goals, we point out the performances that can be gained by using GekkoFS as a burst buffer file system. Finally, we evaluate the data performance of GekkoFS and discuss the measured results.

2.3.1 Experimental Setup

We evaluated the performance of GekkoFS based on various unmodified microbenchmarks which catch access patterns that are common in HPC applications. Our experiments were conducted on the *MOGON II* supercomputer, located at the Johannes Gutenberg University Mainz in Germany. All experiments were performed on Intel 2630v4 Intel Broadwell processors (two sockets each). The node main memory capacity ranges from 64 GiB up to 512 GiB. *MOGON II* uses 100 Gbit/s Intel Omni-Path to establish a fat-tree network between all compute nodes. In addition, each node provides a data center Intel SATA SSD DC S3700 Series as scratch-space (*XFS* formatted) usable within a compute job. We used these SSDs for storing data and metadata of GekkoFS which uses an internal chunk size of 512 KiB. All Lustre experiments were performed on a Lustre scratch file system with 12 Object Storage Targets (OSTs), 2 Object Storage Servers (OSSs), and 1 Metadata Service (MDS) with a total of 1.2 PiB of storage.

Before each experiment iteration, GekkoFS daemons are restarted (requiring less than 20 s for 512 nodes), all SSD content is removed, and kernel buffer, inode, and dentry caches are flushed. The GekkoFS daemon and the application under test are pinned to separate processor sockets to ensure that file system and application do not interfere with each other.

2.3.2 Metadata Performance

We simulated common metadata intensive HPC workloads using the unmodified *mdtest* microbenchmark [41] to evaluate GekkoFS' metadata performance and compare it against a Lustre parallel file system. Although GekkoFS and Lustre have different goals, we point out the performances that can be gained by using GekkoFS as a burst buffer file system. In our experiments, *mdtest* performs *create*, *stat*, and *remove* operations in parallel in a single directory—an important workload in many

Fig. 2 GekkoFS’ file create, stat, and remove throughput for an increasing number of nodes compared to a Lustre file system

HPC applications and among the most difficult workloads for a general-purpose PFS [74].

Each operation on GekkoFS was performed using 100,000 zero-byte files per process (16 processes per node). From the user application’s perspective, all created files are stored within a single directory. However, due to GekkoFS’ internally kept flat namespace, there is conceptually no difference in which directory files are created. This is in contrast to a traditional PFS that may perform better if the workload is distributed among many directories instead of in a single directory.

Figure 2 compares GekkoFS with Lustre in three scenarios with up to 512 nodes: file creation, file stat, and file removal. The y-axis depicts the corresponding operations per second that were achieved for a particular workload on a logarithmic scale. Each experiment was run at least five times with each data point representing the mean of all iterations. GekkoFS’ workload scaled with 100,000 files per process, while Lustre’s workload was fixed to four million files for all experiments. We fixed the number of files for Lustre’s metadata experiments because Lustre was otherwise detecting hanging nodes when scaling to too many files.

Lustre experiments were run in two configurations: All processes operated in a single directory (*single dir*) or each process worked in its own directory (*unique dir*). Moreover, Lustre’s metadata performance was evaluated while the system was accessible by other applications as well.

As seen in Fig. 2, GekkoFS outperforms Lustre by a large margin in all scenarios and shows close to linear scaling, regardless of whether Lustre processes operated in a single or in an isolated directory. Compared to Lustre, GekkoFS achieved around 46 million creates/s ($\sim 1405\times$), 44 million stats/s ($\sim 359\times$), and 22 million removes/s ($\sim 453\times$) on 512 nodes. The standard deviation was less than 3.5% which was computed as the percentage of the mean. Therefore, we achieve our scalability goal, demonstrating the performance benefits of distributing metadata and decoupling directory entries from non-scalable directory blocks (see Sect. 2.2).

Additional GekkoFS experiments were also run while Mogon II was used by other users during production, revealing network interference within the cluster.

With up to 128 nodes we were unable to measure a difference in metadata operation throughput outside of the margin for error compared to the experiments in an undisturbed environment (see Fig. 2). For 256 and 512, we measured a reduced metadata operation throughput between 10 and 20% for create and stat operations. Remove operation throughput remained unaffected.

Lustre’s metadata performance did not scale beyond approximately 32 nodes, demonstrating the aforementioned metadata scalability challenges in such a general-purpose PFS. Moreover, processes in Lustre experiments that operated within their own directory achieved a higher performance in most cases, except for the remove case where Lustre’s `unique dir` remove throughput is reduced by over 70% at 512 nodes compared to Lustre’s `single dir` throughput. This is because the time required to remove the directory of each process (in which it creates its workload) is included in the remove throughput and the number of created `unique` directories increases with the number of used processes in an experiment. Similarly, the time to create the process directories is also included in the create throughput but does not show similar behavior to the case of the remove throughput, indicating optimizations towards create operations.

2.3.3 Data Performance

We used the unmodified *IOR* [31] microbenchmark to evaluate GekkoFS’ I/O performance for sequential and random access patterns in two scenarios: Each process is accessing its own file (file-per-process) and all processes access a single file (shared file). We used 8 KiB, 64 KiB, 1 MiB, and 64 MiB *transfer sizes* to assess the performances for many small I/O accesses and for few large I/O requests. We ran 16 processes on each client, each process writing and reading 4 GiB in total.

GekkoFS data performance is not compared with the Lustre scratch file system as the peak performance of the used Lustre partition, around 12 GiB/s, is already reached for ≤ 10 nodes for sequential I/O patterns. Moreover, Lustre has shown to scale linearly in larger deployments with more OSSs and OSTs being available [48].

Figure 3 shows GekkoFS’ sequential I/O throughput in MiB/s, representing the mean of at least five iterations, for an increasing number of nodes for different transfer sizes. In addition, each data point is compared to the peak performance that all aggregated SSDs could deliver for a given node configuration, visualized as a white rectangle, indicating GekkoFS’ SSD usage efficiency. In general, every result demonstrates GekkoFS’ close to linear scalability, achieving about 141 GiB/s ($\sim 80\%$ of the aggregated SSD peak bandwidth) and 204 GiB/s ($\sim 70\%$ of the aggregated SSD peak bandwidth) for write and read operations for a transfer size of 64 MiB for 512 nodes.

Figure 4 shows GekkoFS’ throughput for random accesses for an increasing number of nodes, showing close to linear scalability in all cases. The file system achieved up to 141 GiB/s write throughput and up to 204 GiB/s read throughput for 64 MiB transfer sizes at 512 nodes.

Fig. 3 GekkoFS' sequential throughput for each process operating on its own file compared to the plain SSD peak throughput. (a) Write throughput. (b) Read throughput

For the file-per-process cases, sequential and random access I/O throughput are similar for transfer sizes larger than the file system's chunk size (512 KiB). This is due to transfer sizes larger than the chunk size internally access whole chunk files while smaller transfer sizes access one chunk at a random offset. Consequently, random accesses for large transfer sizes are conceptually the same as sequential accesses. For smaller transfer sizes, e.g., 8 KiB, random write and read throughput decreased by approximately 33 and 60%, respectively, for 512 nodes owing to the resulting random access to positions within the chunks.

For the shared file cases, a drawback of GekkoFS' synchronous and cache-less design becomes visible. No more than approximately 150K write operations per second were achieved. This was due to network contention on the daemon which maintains the shared file's metadata whose size needs to be constantly updated. To overcome this limitation, we added a rudimentary client cache to locally buffer size updates of a number of write operations before they are sent to the node that manages the file's metadata. As a result, shared file I/O throughput for sequential and random access were similar to file-per-process performances since chunk management on the daemon is then conceptually indifferent in both cases.

Fig. 4 GekkoFS' random throughput for each process operating on its own file. (a) Write throughput. (b) Read throughput

3 Scheduling and Deployment

In order to transfer the data to a previously generated on demand file system in time, the nodes that will be allocated to a job must be known in advance. Today's schedulers plan the resources of a supercomputer. The schedule is based on user requested wall times. Reality shows that the users requested wall times are very inaccurate, and thus the scheduler's predictions are unreliable.

Here two investigations were made and published. In the first work, we have shown that we can improve wall time estimates based on simple job metadata. We also used unconsidered metadata that is usually not publicly available [65].

Predicting the run times of jobs is only one aspect of the challenge. However, the essential factor is the prediction of node allocation to a job. In this second investigation, we have determined the influence of the wall-time on the node prediction [64]. The question we wanted to answer—How good do wall time predictions have to be to predict the allocated nodes accurately?

3.1 Walltime Prediction

One of the challenges is to know which nodes are going to be allocated to a queued job. The HPC scheduler predicts these nodes based on the user given wall times. Therefore, we have decided to evaluate whether there is an easy way to predict such wall time automatically. Our proposed approach for wall time prediction is to train an individual model for every user. For this, we used methods from the machine learning domain and added job metadata, which was in previous work unconsidered. As historical data, we used workloads from two HPC-systems at the Karlsruhe Institute for Technology/Steinbuch Centre for Computing [66], the ForHLR I + II [34, 35] clusters. To train the model, we used automatic machine learning (AUTOML). AUTOML automates the process of hyperparameter optimization and selecting the correct model. We have chosen the auto ML library auto-sklearn [20], which is based on scikit-learn [9, 52].

In Fig. 5 the comparison of the user given wall times and the wall time prediction is shown. As a metric, the median absolute error (medAE) in hours is depicted as cumulative distribution. A model trained with AUTOML shows for 60% of the users a medAE of approximately 1 h on the ForHLR I and 1.4 h for the ForHLR II. The user estimations show a medAE deviation of about 7.4 h on both clusters. So we are able to reduce the median absolute deviation from 7.4 down to 1.4 h in average. Considering the fact that simple methods were used and no insight was provided into the job payload, this result is very good.

Fig. 5 Comparison of median absolute error (medAE) for ForHLR I+II. X-axis Median absolute error in hours, Y-Axis cumulative distribution

3.2 Node Prediction

As mentioned before predicting the run times of jobs is only one aspect of the challenge. However, the decisive factor is the accuracy of node allocation prediction. In this subsequent investigation, we have determined the impact on the node allocation accuracy with improved wall times. Therefore the ALEA Simulator [2] has been extended to simulate the time of the node allocation list [64].

We have conducted several simulations with subsequently improved job run time estimates, from inaccurate wall times as provided by users to fully accurate job run time estimates. For this purpose, we introduce $\tilde{T}_{\text{Req}}$, the “refined” requested wall time,

$$\tilde{T}_{\text{Req}} = T_{\text{Run}} + \lambda(T_{\text{Req}} - T_{\text{Run}}) \quad \text{with} \quad \lambda \in [0, 1], \quad (1)$$

where T_{Req} is the user requested wall time and T_{Run} is the run time of the job. To effectively simulate different precision of requested wall times, each job in the workload is modified by the same λ .

The result of the simulation is shown in Fig. 6, each bar represents a simulation with a different λ value. The bars are categorized into four groups based on the valid node allocation prediction (T_{NAP}). The blue part represents the jobs that are started immediately (instant) execution after the job is submitted to the batch system. These instantly started jobs offer of course no time to create a file system or even stage data. The orange part represents queued jobs with a T_{NAP} between 0 and 1 s. The green part shows jobs with a T_{NAP} from one second up to 10 min and red indicates long term prediction with a valid node allocation prediction over 10 min. The class of jobs with long-term predictions (red) is in our focus. This long-term predictions

Fig. 6 Job distributions of ForHLR II workload with back-filling (CONS). Blue color denotes instant jobs, orange color means job having prediction ≤ 1 s, green color denotes jobs with 1 and 600 s and red color denotes long-term predictions (< 600 s)

increase significantly only at very small $\lambda \leq 0.1$ which proves that very good run time estimates are needed.

3.3 *On Demand Burst Buffer Plugin*

From both evaluations, it is clear, that advanced data staging based on the scheduler prediction is not possible. Also, by using state-of-the-art methods such as machine learning, the accuracy is not sufficient. Therefore we decided to extend the functionality of the SLURM [15] scheduler. Slurm has a feature to manage burst buffers [16]. However, the current implementation status only includes support for the Cray DataWarp solution. Management of burst buffers using other storage technologies is documented, but not yet implemented. With the developed plugin, we extend the functionality of SLURM to create a file system on demand. For the prototype implementation, we also developed tools which deploy BeeOND (BeeGFS On Demand) as an on demand file system per job. Other parallel file systems, e.g. Lustre [7] or GekkoFS, can be added easily. The user requests an on demand file system by a job flag. He can also specify if data should be staged in and out. The SLURM controller marks the jobs and then does the corresponding operations [76].

3.4 *Related Work*

The requested wall times are unfortunately far away from the real used wall time. Gibbons [23, 24], and Downey [19] used historical workloads to predict the wall times of parallel applications. They predict wall times based on templates. These templates are created by analyzing previously collected metadata and grouped according to similarities. However, both approaches are restricted to simple definitions.

In the recent years, machine learning algorithms have been used to predict resource consumption in several studies [33, 40, 43, 44, 61, 70].

Predicting the run-time of jobs is also important in different topics, like for energy aware scheduling [3]. Here the applications' power and performance characteristics are considered to provide an optimized trade off between energy savings and job execution time.

However, all of the above mentioned studies do not try to evaluate the accuracy of the node allocation predictions. Most of the publications focus on observing the utilization of the HPC system and the reliability of the scheduler estimated job start times. In our work, we focus on the node allocation prediction and how good wall time estimates have to be. This directly affects, whether a cross-node, on demand, independent parallel FS can be deployed, and data can be pre-staged, or not.

4 Resource and Topology Detection

Compute nodes of modern HPC systems tend to get more heterogeneous. To plan a proper deployment of the GekkoFS file system on the compute nodes, knowledge of the underlying storage components are vital. This section describes what kind of resource information is of interest and shows possible ways to gather this information. Further, we discuss the architecture of the sysmap tool that we build to collect relevant information.

When thinking about the resources of a compute node, we distinguish between static and dynamic resource usage. Static resource information describes components that do not change frequently and are often similar between nodes. This includes the number of CPU cores, the amount of main memory, the number and capacity of node-local storage devices, or the type of file system. It is unlikely that this kind of hardware is replaced frequently. Otherwise different parts of a Cluster may have different configurations, e.g., one island of a Cluster may have more RAM than another island. On the other hand, dynamic resource usage describes the available resources at a certain point in time.

The goal is to hold a map of the resources available on a system. On the one hand, this can be used as an input for the data staging. On the other hand, such information is useful for the deployment of the file system. When the job scheduler has decided on which set of nodes a job will run, available hardware resources can be queried and an appropriate configuration to deploy the file system can be selected.

The sysmap tool can utilize existing hardware discovery libraries such as *hwloc* [8, 25] by using their interface. While *hwloc* does an excellent job for computing-related artifacts like the number of CPUs or cache sizes, it does not focus on the storage subsystem. Therefore, we use information from the `/proc` and `/sys` pseudo file systems to get information about the system. By reading the system configuration, the sysmap tool gathers information about partitions, mountpoints, file systems but also about available kernel modules and I/O-scheduler configuration. Moreover, we gather network information for InfiniBand networks by utilizing the well-known `ibnetdiscover` tool from the OFED distribution [47].

4.1 Design and Implementation

We have designed an extensible architecture for our sysmap tool. Each resource of interest is captured by a so-called *extractor*. Figure 7 shows a schematic UML-diagram of two extractors. Each extractor module consists of an abstract part, which defines the structure of the data that will be gathered and a specialized part which implements the logic to read the data from a specific source, by overriding the abstract interface. In Fig. 7, the `Filesystem_Extractor` and the `Disk_Extractor` are examples of the abstract parts. The `Linux::Filesystem_Extractor` and the

Fig. 7 Simple UML-Diagram for two example *extractor* modules of our system-map tool

AIX::Filesystem_Extractor are the specialized parts for extracting information of mountpoints and partitions of a specific system [46]. This is useful because the same information may be available on different systems through different sources. On the one hand, the user has to define the abstract extractor he wants, and the sysmap tool selects the source depending on what is available on the target system. On the other hand, we can implement specialized extractor modules for different sources resulting in an equivalent representation of the data for our tool. After gathering the data, the sysmap tool provides a wide variety of output formats presenting the data to the user. Since the tool is mentioned to be executed on multiple compute nodes, the recommended way is to store the results in a central database. Figure 8 depicts an overview of the general workflow of the resource discovery process. The sysmap tool runs on the compute nodes and gathers the resource information. Afterwards, the collected data is stored in a central resource database. For our working prototype, we use a *sqlite*⁵ database. The information can be queried by the sysquery tool, which queries the resource database and outputs the selected data in JSON format. This way the querying component gets a machine-readable section of the required data which can be easily post-processed for their need. Further, the particular database query remains hidden from the user inside the sysquery tool. The datamodel of the resource database is shown in Fig. 9 and consists of four simple tables. The *HostTable* and *ExtractorTable* are to map the hostname or the extractor name to a numerical ID. Information extracted by an extractor is stored as JSON string in the *DataTable*. Further, a *DataID* is maintained to reference the data from an extractor. In the *Host2Data* table, the *DataID* is

⁵<https://www.sqlite.org/index.html>.

Fig. 8 Overview of resource discovery components, the blue components are part of the sysmap tool suite, the resource database is highlighted as the yellow box, the red box represents the querying component, in this case the Job-Scheduler

Fig. 9 The datamodel of the resource database

mapped with the corresponding *HostID*. This way, data that is equal across multiple nodes do not need to be stored multiple times but are easy to query. Since, the output of a query is a JSON string, it makes further processing and output easy for the calling script.

5 On Demand File System in HPC Environment

When using on demand file systems in HPC environments, the premise is that the normal operation should not be affected by the use of on demand file systems. The interference on other jobs should be avoided or even reduced. There should be also no modifications, that have a negative impact on the performance or utilization, of the system.

5.1 Deploying on Demand File System

Usually HPC systems use a *batch system*, such as SLURM [60], MOAB [1], or LSF [30]. The batch system manages the resources of the cluster and starts the user jobs on allocated nodes. Before a job is started, a prologue script may be started on one or all allocated nodes and, if necessary, an epilogue script at the end of a job. These scripts are used to clean, prepare, or test the full functionality of the nodes. We modified these scripts to start the on demand file system upon request. During job submission, a user can request an on demand file system for the job. This solution has minimal impact on the HPC system operation. Users without the need for an on demand file system are not affected. An alternative way of deploying a on demand file system we have described in Sect. 3.3

5.2 Benchmarks

As initial benchmarks we tested the startup and shutdown time of the on demand file system (cf. Table 1). Comparing the startup time of BeeGFS on demand to the startup time of GekkoFS (512 Nodes under 20 s) it is clear, that BeeGFS takes too much time for startup and shutdown at larger scales. BeeGFS has a serial section in its startup where a status file is created on every node sequentially. This was also discussed on the mailing list [63] with a possible solution to improve the behavior in future releases.

In Fig. 10 we show the IoZone [10] benchmark to measure the read and write throughput of the on demand file system (solid line). The figure shows that performance increases linearly with the number of used compute nodes. The limiting factor here is the aggregate throughput of the used SATA-SSDs. A small throughput variation can be observed due to normal performance scattering of SSDs [36]. The dotted line indicates the theoretical throughput with NVMe devices. Here we assumed the performance for today's common PCIe $\times 4$ NVM devices [29] with a throughput of 3500/2000 MB/s of read/write performance.

In a further test, we evaluated the storage pooling feature of BeeGFS [4]. We created a storage pool for each switch according to the network topology. In other words, when writing to a storage pool, the data is distributed via the stripe count and chunk size, but remains within the storage pool and thus on a switch. Figure 11 shows the write throughput for three scenarios. Each scenario uses a different number of core switches with six being the full network capacity. In the first

Table 1 BeeGFS startup and shutdown

Nodes	8	16	32	64	128	256
Startup (s)	10.2	16.7	29.3	56.5	152.1	222.4
Shutdown (s)	11.9	12.1	9.4	15.9	36.1	81.0

Fig. 10 Solid line: Read/write throughput. Dashed line: extrapolation with the theoretical peak of NVMe-SSDs

Fig. 11 IoZone write throughput with reduced number of core switches on 240 nodes

experiment, with all six core switches, there is only a minimal performance loss, which indicates a small overhead when using storage pools. In the second case we turned off three switches, and in the last case we turned off five switches. With reduced number of core switches the write throughput drops due to the reduced network capacity. If storage pools are created according to the topology, it is possible to achieve the same performance as with all six switches.

5.3 Concurrent Data Staging

We also considered the case of copying data back to the PFS while an application is running. For this purpose, we evaluated NASTJA [6] with concurrent data staging. To stage the data, during the NASTJA execution, we used the parallel copy tool dcp [59]. The configuration for this use-case:

- We used 24 nodes with 20 cores per node.
- NASTJA was executed on 23 nodes with 20 tasks per node.
- BeeOND was started on all 24 nodes using the idle node as metadata server.
- Three different scenarios were evaluated during the application execution:
 - without data staging,
 - data staging using every node with one task per node for data staging,
 - data staging using the node, where only the meta-data server is running, with 4 tasks executed on this node.

Figure 12 shows the average execution time per time-step of five runs in our different scenarios. In the beginning, the slowdown is significant (orange line) due to the high amount of metadata operations. In this case, a portion of the data is indexed on every node. This indexing is causing interference with the application. When using only the MDS-server to copy the data (green line), the indexing is done only on the MDS-server.

Fig. 12 Average execution time per time-step (5 runs). Without data staging(blue). Concurrent data staging using the meta-data node (green) and using every node (orange)

6 GekkoFS on NVME Based Storage Systems

Recently, new storage technologies such as NVME SSDs have been introduced to modern HPC systems. To evaluate the GekkoFS file system, for future systems, we performed some benchmarks using NVME SSDs. For demonstration we installed GekkoFS on the cluster Taurus [67] of TU Dresden. Taurus consists of ca. 47,000 cores of different architectures. For the demonstration, we use 8 NVME nodes of the HPC-DA [28] extension of Taurus. This extension consists of 90 nodes, and a single node has 8 Intel SSD DC P4610 Series NVME storage with 3.2 TB capacity and a peak bandwidth of 3.2 GB/s. Each node has 2 sockets Intel Xeon E5-2620 v4 with 32 cores and 64 GB main memory. Further, the NVME nodes are equipped with two 100 Gbit/s EDR Infiniband links with a peak bandwidth of 25 GB/s each. This experiment aims to investigate how well GekkoFS performs on new storage architectures.

We installed GekkoFS on Taurus using the Infiniband network provider. For our demonstration, we use 8 NVME nodes in this setup. The nodes are client and server in one. We assign one NVME card per node as backing storage to the GekkoFS daemon. This results in a distributed file system with a total capacity of 25.6 TB and a theoretical maximum bandwidth of $8 \times 3.2 \text{ GB/s} = 25.6 \text{ GB/s}$ for this configuration. To measure the data throughput of GekkoFS and investigate the impact of different access patterns to the file system we utilize the IOR benchmark.

We perform strong scaling tests with 8, 16, 32 and 64 processes writing and reading 1 TB of data. Therefore, we adjust the block size and transfer size for a different number of processes. To avoid interference, we pin the IOR processes to one socket while the GekkoFS daemon is pinned to the other. Before the creation of the GekkoFS file system the NVME devices were cleared, and a new Ext4 file system was created as an underlying file system on the block device. We measure different access patterns, file per process with sequential and random accesses and shared file with sequential access. To avoid measuring cache effects, we flush the page, inode and dentry caches of the operating system before each run.

Figure 13 shows the sequential access pattern. In the figure, one can see that the write bandwidth is stable at around 22 GB/s for all runs. The variation is small, and the values are near to the peak bandwidth of 25 GB/s for this setup. The suitable write bandwidths came from the relatively large transfer sizes of 64 MB to benefit from RDMA. For the read bandwidth, we get values between 13 and 17 GB/s. Also the read bandwidth first decreases when more processes are used and then increases again at the 64 processes. Such a poor read bandwidth is a behavior which could not be observed for the other measurements on MOGON II, where read and write bandwidth are almost equal, and is certainly a point of further investigation.

Figure 14 depicts the random access case. The results are similar to the sequential access pattern, which was expected because the internal handling of GekkoFS makes no difference for these cases. The write bandwidth is stable between 22 and 23 GB/s

Fig. 13 IOR on GekkoFS on 8 NVME nodes performing a sequential file per process access pattern

and saturates the NVME SSD quite well. For the read, the achieved bandwidth is around 14 GB/s, the values are more stable than for the sequential case, which might be some cache effects.

In Fig. 15 we can see, that even for shared access pattern the results are similar to the file per process access pattern. The write bandwidth is again stable at 22 GB/s and the read bandwidth is around 16 GB/s except for the configuration with 32 processes where the read bandwidth is lower. This is also similar to the sequential file per process configuration in Fig. 13. As a result, we can see that GekkoFS can utilize NVME SSDs and is, therefore, ready for the next generation of storage systems. We could figure out that the different access patterns make no difference for the write bandwidth. For the read bandwidth, there is some bottleneck which needs further investigation. At the time of writing, multiple causes are imaginable; for example, the network layer for Infiniband might be an issue. This could also explain why this problem does not occur for the tests in Mainz because they have other network types.

Fig. 14 IOR on GekkoFS on 8 NVME nodes performing a random file per process access pattern

Fig. 15 IOR on GekkoFS on 8 NVME nodes performing a sequential shared file access pattern

7 Conclusion

The goal of the ADA-FS project was to improve I/O performance for parallel applications. Therefore, a distributed burst buffer file system, and several components for deployment and data management were developed. The GekkoFS distributed burst buffer file system as the central part of the project was presented as a scalable and very flexible alternative to handle the challenging I/O patterns of scientific applications. Primarily through the innovative metadata management, it beats conservative shared parallel file systems for metadata intensive workloads by a margin. Thanks to its flexibility, GekkoFS offers the user an exclusive file system for his applications and eliminates several bottlenecks caused by the contention of a shared resource. In addition, GekkoFS has become a basis in the EU-funded *Next Generation I/O for Exascale* (NEXTGenIO) project where it will be continuously and collaboratively developed to support future storage technologies as well, such as persistent memory.

For successful data staging, investigations about the precision of the user-provided wallclock time of jobs were made. We could show how to improve wallclock estimates by considering the metadata of a job, and show a way to integrate the process of deployment and data staging into the job scheduler. Further, we present a tool suite to collect information about hardware resources of a compute node to support the deployment in a flexible manner.

Another topic that was not covered here is the analysis of the required POSIX semantics of parallel applications. These insights show during the design of the file system which operations are required to run scientific workloads. Further, its results can help the user to decide for a storage system that fits his needs best.

The evaluations showed that GekkoFS provides close to linear data and metadata scalability up to 512 nodes with tens of millions of metadata operations. Due to the decentralized and distributed design, the file system is set to be used in even larger environments as exascale environments are in close reach. Even on the latest storage infrastructure, GekkoFS can operate out of the box at the peak bandwidth at least for write operations.

Following this project, we plan further improvements on GekkoFS, for example, caching offers possibilities to gain even more performance. Another topic that we want to keep working on is the integration of GekkoFS into the job schedulers of the systems and the workflows of the user.

Conclusively, the project reached its goals by improving I/O performance of parallel applications, especially in the field of metadata intensive workloads where traditional parallel file systems are lacking performance.

Acknowledgments This work as part of the project ADA-FS is funded by the DFG Priority Program “Software for Exascale Computing” (SPPEXA, SPP 1648), which is gratefully acknowledged.

This research was conducted using the supercomputer ForHLR II and services offered by Karlsruher Institute of Technology and the Steinbuch Centre for Computing. The authors gratefully acknowledge the time and granted access to ForHLR II.

Parts of this research were conducted using the supercomputer Mogon II and services offered by Johannes Gutenberg University Mainz. The authors gratefully acknowledge the computing time granted on Mogon II.

The authors gratefully acknowledge the GWK support for funding this project by providing computing time through the Center for Information Services and HPC (ZIH) at TU Dresden on the HPC-DA.

References

1. Adaptive Computing. <http://www.adaptivecomputing.com>
2. Alea 4: Job scheduling simulator (2019). <https://github.com/aleasimulator>
3. Auweter, A., Bode, A., Brehm, M., Brochard, L., Hammer, N., Huber, H., Panda, R., Thomas, F., Wilde, T.: A case study of energy aware scheduling on supermuc. In: Proceedings of the 29th International Conference on Supercomputing, ISC 2014, vol. 8488, pp. 394–409. Springer, New York (2014). https://doi.org/10.1007/978-3-319-07518-1_25
4. BeeGFS: BeeGFS Storage Pool. <https://www.beegfs.io/wiki/StoragePools> (2018). Accessed: August 1 201
5. Bent, J., Gibson, G.A., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: PLFS: a checkpoint filesystem for parallel applications. In: Proceedings of the ACM/IEEE Conference on High Performance Computing (SC), November 14–20, Portland, (2009)
6. Berghoff, M., Kondov, I., Hötzer, J.: Massively parallel stencil code solver with autonomous adaptive block distribution. *IEEE Trans. Parallel Distrib. Syst.* **29**(10), 2282–2296 (2018). <https://doi.org/10.1109/TPDS.2018.2819672>
7. Braam, P.J., Schwan, P.: Lustre: The intergalactic file system. In: Ottawa Linux Symposium, p. 50 (2002)
8. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in HPC applications. In: Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing, PDP 2010 (2010)
9. Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., VanderPlas, J., Joly, A., Holt, B., Varoquaux, G.: API design for machine learning software: experiences from the scikit-learn project. In: ECML PKDD Workshop: Languages for Data Mining and Machine Learning, pp. 108–122 (2013)
10. Capps, D., Norcott, W.: Iozone filesystem benchmark (2008). <http://iozone.org/>
11. Carns, P., Yao, Y., Harms, K., Latham, R., Ross, R., Antypas, K.: Production I/O characterization on the Cray XE6. In: Proceedings of the Cray User Group meeting, vol. 2013 (2013)
12. Carns, P.H., III, W.B.L., Ross, R.B., Thakur, R.: PVFS: A parallel file system for Linux clusters. In: 4th Annual Linux Showcase and Conference 2000. Atlanta (2000)
13. Carns, P.H., Jenkins, J., Cranor, C.D., Atchley, S., Seo, S., Snyder, S., Ross, R.B.: Enabling NVM for data-intensive scientific services. In: 4th Workshop on Interactions of NVM/Flash with Operating Systems and Workloads, INFLOW@OSDI 2016. Savannah (2016)
14. Crandall, P., Aydt, R.A., Chien, A.A., Reed, D.A.: Input/output characteristics of scalable parallel applications. In: Proceedings Supercomputing '95, San Diego, p. 59 (1995)
15. Documentation, S.: Slurm workload manager—overview. <https://slurm.schedmd.com/overview.html>
16. Documentation, S.: Slurm workload manager—slurm burst buffer guide. https://slurm.schedmd.com/burst_buffer.html

17. Dong, S., Callaghan, M., Galanis, L., Borthakur, D., Savor, T., Strum, M.: Optimizing space amplification in rocksdb. In: Proceedings of the 8th Biennial Conference on Innovative Data Systems Research, CIDR 2017. Chaminade (2017)
18. Dorier, M., Antoniu, G., Ross, R.B., Kimpe, D., Ibrahim, S.: Calciom: Mitigating I/O interference in HPC systems through cross-application coordination. In: Processing Symposium on 2014 IEEE 28th International Parallel and Distributed, pp. 155–164. Phoenix (2014)
19. Downey, A.B.: Predicting queue times on space-sharing parallel computers. In: Proceedings of the 11th International Parallel Processing Symposium, pp. 209–218. IEEE, Piscataway (1997)
20. Feurer, M., Klein, A., Eggenberger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (eds.) *Advances in Neural Information Processing Systems*, vol. 28, pp. 2962–2970. Curran Associates, Red Hook (2015). <http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf>
21. Folk, M., Cheng, A., Yates, K.: HDF5: a file format and I/O library for high performance computing applications. In: *Proceedings of the Supercomputing*, vol. 99, pp. 5–33 (1999)
22. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*, Portland (2009)
23. Gibbons, R.: A historical application profiler for use by parallel schedulers. In: *Job Scheduling Strategies for Parallel Processing*, pp. 58–77. Springer, Berlin (1997)
24. Gibbons, R.: A historical profiler for use by parallel schedulers. Master's thesis, University of Toronto (1997)
25. Goglin, B.: Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc). In: *2014 International Conference on High Performance Computing and Simulation (HPCS)*, pp. 74–81. IEEE, Piscataway (2014)
26. Herold, F., Breuner, S., Heichler, J.: An introduction to beegfs (2014). https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf
27. Hey, T., Tansley, S., Tolle, K.M. (eds.): *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research (2009)
28. HPC for Data Analytics (2019). <https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/HPCDA>. Accessed 25 July 2019
29. Hung, J.J., Bu, K., Sun, Z.L., Diao, J.T., Liu, J.B.: PCI express-based NVMe solid state disk. In: *Applied Mechanics and Materials*, vol. 464, pp. 365–368. Trans Tech Publications (2014)
30. IBM—platform computing. <http://www.ibm.com/systems/platformcomputing/products/lfs/>
31. Ior data benchmark (2018). <https://github.com/hpc/ior>
32. Jasak, H., Jemcov, A., Tukovic, Z., et al.: Openfoam: a C++ library for complex physics simulations. In: *International Workshop on Coupled Methods in Numerical Dynamics*, vol. 1000, pp. 1–20 (2007)
33. Kapadia, N.H., Fortes, J.A.: On the design of a demand-based network-computing system: The purdue university network-computing hubs. In: *Proceedings of the Seventh International Symposium on High Performance Distributed Computing*, pp. 71–80. IEEE, Piscataway (1998)
34. Forschungshochleistungsrechner ForHLR 1 (2018). www.scc.kit.edu/dienste/forhlr1.php
35. Forschungshochleistungsrechner ForHLR 2 (2018). www.scc.kit.edu/dienste/forhlr2.php
36. Kim, E.: SSD performance-a primer. In: *Solid State Storage Initiative* (2013)
37. Lensing, P.H., Cortes, T., Hughes, J., Brinkmann, A.: File system scalability with highly decentralized metadata on independent storage devices. In: *IEEE/ACM 16th International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, Cartagena, pp. 366–375 (2016)
38. Liu, N., Cope, J., Carns, P.H., Carothers, C.D., Ross, R.B., Grider, G., Crume, A., Maltzahn, C.: On the role of burst buffers in leadership-class storage systems. In: *IEEE 28th Symposium on Mass Storage Systems and Technologies, MSST 2012*, Pacific Grove, pp. 1–11 (2012)
39. Lofstead, J.F., Klasky, S., Schwan, K., Podhorszki, N., Jin, C.: Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In: *Proceedings of the Sixth International Workshop on Challenges of Large Applications in Distributed Environments, CLADE@HPDC 2008*, Boston, pp. 15–24 (2008)

40. Matsunaga, A., Fortes, J.A.: On the use of machine learning to predict the time and resources consumed by applications. In: *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 495–504. IEEE Computer Society, Washington (2010)
41. Mdttest metadata benchmark (2018). <https://github.com/hpc/ior>
42. Moore, M., Bonnie, D., Ligon, B., Marshall, M., Ligon, W., Mills, N., Quarles, E., Sampson, S., Yang, S., Wilson, B.: OrangeFS: Advancing PVFs. FAST poster session (2011)
43. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel Distrib. Syst.* **12**(6), 529–543 (2001)
44. Nadeem, F., Fahringer, T.: Using templates to predict execution time of scientific workflow applications in the grid. In: *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 316–323. IEEE Computer Society, Washington (2009)
45. Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C.S., Best, M.L.: File-access characteristics of parallel scientific workloads. *IEEE Trans. Parallel Distrib. Syst.* **7**(10), 1075–1089 (1996)
46. Oeste, S., Kluge, M., Soysal, M., Streit, A., Vef, M., Brinkmann, A.: Exploring opportunities for job-temporal file systems with ADA-FS. In: *Proceedings of the First Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems* (2016)
47. OpenFabrics Alliance: Open fabric enterprise distribution (2016). <https://www.openfabrics.org/>. Accessed 16 July 2016
48. Oral, S., Dillow, D.A., Fuller, D., Hill, J., Leverman, D., Vazhkudai, S.S., Wang, F., Kim, Y., Rogers, J., Simmons, J., et al.: OLCFs 1 TB/s, next-generation lustre file system. In: *Proceedings of the Cray User Group Conference (CUG 2013)*, pp. 1–12 (2013)
49. Oral, S., Shah, G.: Spectrum scale enhancements for coral. Paper presentation slides at supercomputing'16. (2016). http://files.gpfsug.org/presentations/2016/SC16/11_Sarp_Oral_Gautam_Shah_Spectrum_Scale_Enhancements_for_CORAL_v2.pdf
50. Patil, S., Gibson, G.A.: Scale and concurrency of GIGA+: file system directories with millions of files. In: *Proceedings of the Ninth USENIX Conference on File and Storage Technologies*, San Jose, pp. 177–190 (2011)
51. Patil, S., Ren, K., Gibson, G.: A case for scaling HPC metadata performance through de-specialization. In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Salt Lake City, pp. 30–35 (2012)
52. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. *J. Mach. Learn. Res.* **12**, 2825–2830 (2011)
53. Qian, Y., Li, X., Ihara, S., Zeng, L., Kaiser, J., Süß, T., Brinkmann, A.: A configurable rule based classful token bucket filter network request scheduler for the lustre file system. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, Denver, pp. 6:1–6:12 (2017)
54. Ren, K., Zheng, Q., Patil, S., Gibson, G.A.: IndexFS: scaling file system metadata performance with stateless caching and bulk insertion. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014*, New Orleans, pp. 237–248 (2014)
55. Ross, R., Thakur, R., Choudhary, A.: Achievements and challenges for I/O in computational science. In: *Journal of Physics: Conference Series*, vol. 16, p. 501 (2005)
56. Ross, R.B., Latham, R.: PVFS—PVFS: a parallel file system. In: *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, Tampa, p. 34 (2006)
57. Schmuck, F.B., Haskin, R.L.: GPFS: a shared-disk file system for large computing clusters. In: *Proceedings of the FAST '02 Conference on File and Storage Technologies*, Monterey, pp. 231–244 (2002)
58. Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., Carns, P.H., Castelló, A., Genet, D., Hérault, T., Iwasaki, S., Jindal, P., Kalé, L.V., Krishnamoorthy, S., Lifflander, J., Lu, H., Meneses, E., Snir, M., Sun, Y., Taura, K., Beckman, P.H.: Argobots: A lightweight low-level threading and tasking framework. *IEEE Trans. Parallel Distrib. Syst.* **29**(3), 512–526 (2018)

59. Sikich, D., Di Natale, G., LeGendre, M., Moody, A.: mpiFileUtils: A Parallel and Distributed Toolset for Managing Large Datasets. Tech. rep., Lawrence Livermore National Lab.(LLNL), Livermore (2017)
60. Slurm—schedmd. <http://www.schedmd.com>
61. Smith, W.: Prediction services for distributed computing. In: IEEE International Parallel and Distributed Processing Symposium, pp. 1–10. IEEE, Piscataway (2007)
62. Soumagne, J., Kimpe, D., Zounmevo, J.A., Chaarawi, M., Koziol, Q., Afsahi, A., Ross, R.B.: Mercury: enabling remote procedure call for high-performance computing. In: 2013 IEEE International Conference on Cluster Computing, CLUSTER 2013, Indianapolis, pp. 1–8 (2013)
63. Soysal, M.: fhfgs mailing list discussion (2017). <https://groups.google.com/d/msg/fhfgs-user/g8ysFS35Ucs/E-RtxKyiCAAJ>
64. Soysal, M., Berghoff, M., Klusáček, D., Streit, A.: On the quality of wall time estimates for resource allocation prediction. In: Proceedings of the 48th International Conference on Parallel Processing: Workshops, ICPP 2019, pp. 23:1–23:8. ACM, New York, (2019). <https://doi.org/10.1145/3339186.3339204>
65. Soysal, M., Berghoff, M., Streit, A.: Analysis of job metadata for enhanced wall time prediction. In: Proceedings of the 22nd International Workshop on Job Scheduling Strategies for Parallel Processing, JSSPP 2018, Vancouver. Revised selected papers, Klusáček, D (ed.) Lecture Notes in Computer Science, vol. 11332, p. 14 S. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-10632-4_1. 46.11.01; LK 01
66. Steinbuch Center for Computing (SCC). <http://www.scc.kit.edu> (2016). Accessed 16 August 2016
67. Bull HPC-Cluster Taurus (2019). <https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemTaurus>. Accessed 25 July 2019
68. Thapaliya, S., Bangalore, P., Lofstead, J.F., Mohror, K., Moody, A.: Managing I/O interference in a shared burst buffer system. In: 45th International Conference on Parallel Processing, ICPP 2016, Philadelphia, pp. 416–425 (2016)
69. Thinkparq, BeeGFS: Beegfs the leading parallel cluster file system (2018). https://www.beegfs.io/docs/BeeGFS_Flyer.pdf
70. Tsafirir, D., Etsion, Y., Feitelson, D.G.: Backfilling using system-generated predictions rather than user runtime estimates. IEEE Trans. Parallel Distrib. Syst. **18**(6), 789–803 (2007)
71. Vef, M., Moti, N., Süß, T., Tacke, M., Tocci, T., Nou, R., Miranda, A., Cortes, T., Brinkmann, A.: Gekkofs—a temporary burst buffer file system for HPC applications. J. Comput. Sci. Technol. **35**(1), 72–91 (2020)
72. Vef, M., Moti, N., Süß, T., Tocci, T., Nou, R., Miranda, A., Cortes, T., Brinkmann, A.: Gekkofs—a temporary distributed file system for HPC applications. In: IEEE International Conference on Cluster Computing, CLUSTER 2018, Belfast, pp. 319–324 (2018)
73. Vef, M.A.: Analyzing file create performance in IBM spectrum scale. Master's thesis, Johannes Gutenberg University Mainz (2016). <http://www.staff.uni-mainz.de/vef/pubs/vef2016thesis.pdf>
74. Vef, M.A., Tarasov, V., Hildebrand, D., Brinkmann, A.: Challenges and solutions for tracing storage systems: a case study with spectrum scale. ACM Trans. Storage **14**(2), 18:1–18:24 (2018)
75. Vilayannur, M., Nath, P., Sivasubramaniam, A.: Providing tunable consistency for a parallel file store. In: Proceedings of the FAST '05 Conference on File and Storage Technologies, San Francisco (2005)
76. Voigt, V.: Entwicklung eines on-demand burst-buffer-plugins fuer HPC-batch-systeme
77. Wang, F., Xin, Q., Hong, B., Brandt, S.A., Miller, E., Long, D., McLarty, T.: File system workload analysis for large scale scientific computing applications. Tech. rep., Lawrence Livermore National Laboratory (LLNL), Livermore (2004)
78. Wang, T., Mohror, K., Moody, A., Sato, K., Yu, W.: An ephemeral burst-buffer file system for scientific applications. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, pp. 807–818 (2016)

79. Xing, J., Xiong, J., Sun, N., Ma, J.: Adaptive and scalable metadata management to support a trillion files. In: Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, Portland (2009)
80. Yang, S., Ligon III, W.B., Quarles, E.C.: Scalable distributed directory implementation on orange file system. In: Proceedings of the IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI) (2011)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.