Dynamically Adaptable I/O Semantics for High Performance Computing

Michael Kuhn(✉)

University of Hamburg, Hamburg, Germany
michael.kuhn@informatik.uni-hamburg.de

Abstract. While an input/output (I/O) interface's syntax describes the available operations, its semantics determines how these operations behave and which assumptions developers can make about them. There are several different interface standards in existence, some of them dating back decades and having been designed for local file systems; one such representative is POSIX. Many parallel distributed file systems implement a POSIX-compliant interface to improve portability. All currently available interfaces follow a fixed approach regarding semantics, making them only suitable for a subset of use cases and workloads. While the interfaces do not allow application developers to influence the I/O semantics, applications could benefit greatly from the possibility of being able to adapt them to their requirements.

The work presented in this paper includes the design of a novel I/O interface and a file system called JULEA. They offer support for dynamically adaptable semantics and are suited specifically for HPC applications. The introduced concept allows applications to adapt the file system behavior to their exact I/O requirements instead of the other way around. The general goal is an interface that allows developers to specify *what* operations should do and *how* they should behave – leaving the actual realization and possible optimizations to the underlying file system.

JULEA has been evaluated using both synthetic benchmarks and real-world applications. Overall, JULEA provides data and metadata performance comparable to that of other established parallel distributed file systems. However, in contrast to the existing solutions, its flexible semantics allows it to cover a wider range of use cases in an efficient way. The results demonstrate that there is need for I/O interfaces that can adapt to the requirements of applications. Even though POSIX facilitates portability, it does not seem to be suited for contemporary HPC demands.

Keywords: I/O semantics · I/O interface · Parallel file system

1 Introduction

Throughout their history, the computational power of supercomputers has been increasing exponentially, doubling roughly every 14 months [20]. While this computational power has allowed more accurate simulations to be performed, this

J.M. Kunkel and T. Ludwig (Eds.): ISC High Performance 2015, LNCS 9137, pp. 240–256, 2015.
DOI: 10.1007/978-3-319-20119-1_18

has also caused the simulation results to grow in size. Due to the large amounts of data produced by parallel applications, high performance I/O is an important aspect because storing and retrieving such large amounts of data can greatly affect the overall performance of these applications.

However, the I/O requirements of parallel applications can vary widely: While some applications process large amounts of input data and produce relatively small results, others might work using a small set of input data and output large amounts of data; additionally, the aforementioned data can be spread across many small files or be concentrated into few large files. Naturally, any combination thereof is also possible. These different requirements can make high demands on supercomputers' storage systems.

Parallel distributed file systems provide one or more I/O interfaces that can be used to access data within the file system. Usually at least one of them is standardized, while additional proprietary interfaces might offer improved performance at the cost of portability. Popular interface choices include POSIX [9], MPI-IO [16], HDF [19] and NetCDF [18]. Almost all the I/O interfaces found in HPC today offer simple byte- or element-oriented access to data and thus do not have any a priori information about what kind of data the applications access and how the high-level access patterns look like. However, this information can be very beneficial for optimizing the performance of I/O operations.

While the I/O interface defines which I/O operations are available, the I/O semantics describes and defines the behavior of these operations. Usually each I/O interface is accompanied by a set of I/O semantics, tailored to this specific interface. The POSIX I/O semantics is probably both the oldest and the most widely used semantics, even in HPC. However, due to being designed for traditional local file systems, it imposes unnecessary restrictions on today's parallel distributed file systems. POSIX's very strict consistency requirements that require write operations to be propagated to all other clients immediately are one of these restrictions and can lead to performance bottlenecks in distributed environments [21]. Parallel distributed file systems often implement the strict POSIX I/O semantics to accommodate applications that require it or simply expect it to be available for portability reasons. However, this can lead to suboptimal behavior for many use cases because its strictness is often not necessary. Even though application developers usually know their applications' requirements and could easily specify them for improved performance, current I/O interfaces and file systems do not provide appropriate facilities for this task.

Performing I/O efficiently is becoming an increasingly important problem. CPU speed and HDD capacity have roughly increased by factors of 500 and 100 every 10 years, respectively [20,24]. The speed of HDDs, however, has only grown by a factor of 10 every 10 years [23]; even newer technologies such as SSDs only offer a factor of 18. Although it is theoretically possible to compensate for this fact in the short term by simply buying more storage hardware, the ever increasing gap between the exponentially growing processing power on the one hand and the stagnating storage capacity and throughput on the other hand, requires new approaches to use the storage infrastructure as efficiently as possible.

The goal of this paper is to explore the usefulness of additional semantical information in the I/O interface. The JULEA framework introduces a newly designed I/O interface featuring dynamically adaptable semantics that is suited specifically for HPC applications. It allows application developers to specify the semantics of I/O operations at runtime and supports batch operations to increase performance. The overall goal is to allow the application developer to specify the desired behavior and leave the actual realization to the I/O system.

This paper is structured as follows: The most important design aspects of the JULEA I/O interface are elaborated in Sect. 2, focusing on the differences to traditional I/O interfaces and file systems. Section 3 covers related work and compares JULEA's design with existing approaches. Section 4 contains an analysis of the behavior of different file systems using synthetic benchmarks. A conclusion and future work are given in Sect. 5.

2 Interface and File System Design

JULEA's general architecture closely follows that of established parallel distributed file systems such as Lustre [5] and OrangeFS [4]. Machines can have one or several of three different roles: client, data server or metadata server. While it is possible to have a machine perform all three roles simultaneously, it is recommended to separate the clients from the servers to provide stable performance. JULEA supports multiple data and metadata servers and allows data and metadata to be distributed among them; it is possible to influence the actual distribution of data using distributions.

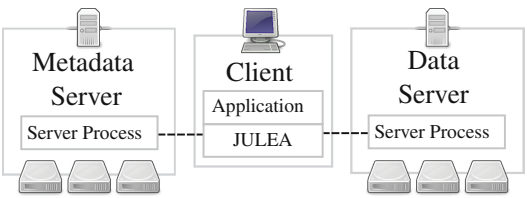
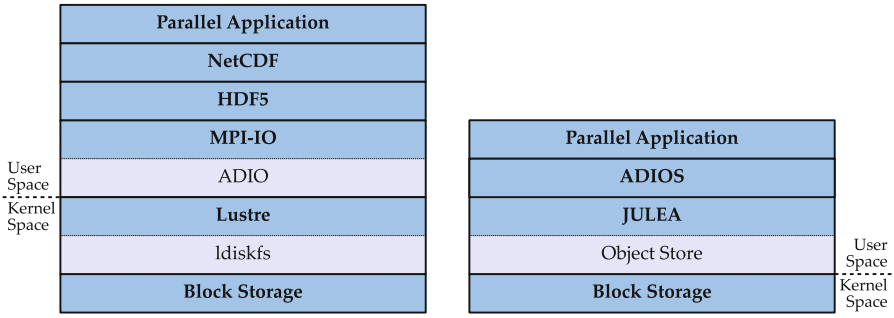


Fig. 1. JULEA's file system components

A very brief general view of JULEA's different components and their interactions with each other are shown in Fig. 1. Applications are able to use JULEA's I/O interface that talks directly to the data and metadata servers by linking against its client library `libjulea.so`; it abstracts all the internal details and provides a convenient interface for developers. The metadata and data servers run on dedicated machines with attached storage hardware. While the metadata servers make use of the MongoDB database system, the data servers run a user space daemon called `julea-daemon` that handles all I/O on behalf of the clients.

Figures 2a and b show a comparison of an exemplary HPC I/O stack and the proposed JULEA I/O stack. In addition to the logical layers, the separation



(a) HPC I/O stack (b) JULEA I/O stack

Fig. 2. Current HPC I/O stack and proposed JULEA I/O stack

between kernel and user space is shown. All kernel space layers are either implemented directly inside the kernel or as kernel modules; the user space layers are either normal applications or libraries. As can be seen, JULEA's architecture features less layers, which makes it easier to analyze the actual I/O behavior of applications. It also allows concentrating all optimizations into a single layer, reducing the implementation and runtime overhead.

The current I/O stack's design results in several transformations of the data as it is being transported through the different layers. The parallel application's data types are stored in NetCDF that in turn stores its data in HDF's datasets and groups. This data is then transformed into a byte stream for MPI-IO. It then stores the data in the actual parallel distributed file system that splits up the data and stripes it across its servers, potentially storing it in yet another underlying local file system.

An important design goal of JULEA is to remove the duplication of functionality found in the traditional HPC I/O stack. Because many distributed file systems use an underlying local POSIX file system to store the actual data and metadata, a lot of common file system functionality such as path lookup and permission checking is duplicated. This can be achieved by completely eliminating the underlying POSIX file systems and using suitable object stores.

Because it is often unreasonable to port applications to new and experimental I/O interfaces due to their size and complexity, it makes sense to leverage a layer providing compatibility for existing applications. ADIOS is an established I/O interface and specifically allows implementing different backends. To minimize the overhead, ADIOS could be used as a relatively thin layer on top of JULEA to provide convenient access for application developers.

2.1 File System Namespace

Traditional file systems allow deeply nested directory structures. To avoid the overhead caused by this, only a restricted and relatively flat hierarchical namespace is

supported. While this approach might be unsuited for a general purpose file system, JULEA is explicitly focused on specific use cases that are commonly found in HPC. Therefore, JULEA is meant to be used in conjunction with traditional file systems like NFS to provide other parts of the infrastructure such as the users' home directories.

The file system namespace is divided into *stores*, *collections*, and *items*. Each store can contain multiple collections that can, in turn, contain multiple items. In traditional POSIX file systems, each component of the potentially deeply nested path has to be checked for each access. This can seriously hamper performance, especially for distributed file systems.

2.2 Interface

JULEA's interface has been designed from scratch to offer simplicity of use while still meeting the requirements of high performance and dynamically adaptable semantics. Its functionality can be subdivided into five groups:

1. **Batches:** Multiple operations can be batched explicitly to improve performance by reducing network overhead.
2. **Distributions:** It is possible to influence the distribution of data directly to optimize its placement on the data servers if necessary.
3. **Namespace:** The file system namespace is accessible using a convenient abstraction called uniform resource identifiers (URIs).
4. **Semantics:** JULEA's semantics is dynamically adaptable according to the applications' I/O requirements.
5. **Stores, Collections and Items:** It is possible to create, remove, open and iterate over all of JULEA's file system objects.

All of the above functionality is available publicly and directly to developers. The two most important features are the ability to specify semantical information and to batch operations. Both approaches give the file system additional information that can be used to optimize accesses. Due to their importance, these two features will be explained in more detail; more information about the other ones can be found in [13].

It is possible for developers and users to specify additional information equivalent to the coarse-grained statement "this is a checkpoint" or the more fine-grained "this operation requires strict consistency semantics". This allows the file system to tune operations for specific applications by itself. Additionally, developers are able to emulate well-established semantics as well as mixing different semantics within one application.

Developers perform all accesses to the file systems via so-called *batches*. Each batch can consist of multiple operations. It is also possible to combine different kinds of operations within one batch. For instance, one batch might create a collection and several items within it, and write data to each of the items. Because the file system has knowledge about all operations within one batch, more elaborate optimizations can be performed.

Traditional POSIX file systems can also try to aggregate multiple operations to improve network utilization. However, this can only be done by caching these operations in the client's main memory for a given amount of time and then performing these optimizations. Because the POSIX interface does not provide enough information to make reliable decisions for these kinds of optimizations, it is necessary to employ heuristics, resulting in suboptimal behavior for borderline cases. Additionally, it is not possible to do this in all cases because it would violate the POSIX semantics. Therefore, users can never be sure when exactly operations are performed in such a system without calling synchronization functions explicitly, which can be very expensive.[1]

To be able to easily overlap calculations and I/O, it is possible to execute batches asynchronously. This support is offered natively by the I/O interface without forcing developers to resort to using background threads or similar techniques. The file system also exports additional information to enable performance optimizations such as aligning data to the file system's stripe size, which is crucial for high performance [2].

2.3 Semantics

JULEA allows many aspects of the file system operations' semantics to be changed at runtime. Several key areas of the semantics have been identified as important to provide opportunities for optimizations: atomicity, concurrency, consistency, ordering, persistency and safety. Even though it is possible to mix the settings for each of these semantics, not all combinations produce reasonable results.

The *atomicity* semantics can be used to specify whether accesses should be executed atomically, that is, whether or not it is possible for clients to see intermediate states of operations. If atomicity is required, some kind of locking has to be performed to prevent other clients from accessing data that is currently being modified. The atomicity semantics is clearly performance-related. Atomic accesses operating on the same data have to be serialized, which implies a performance penalty. If atomicity is not required, all operations can be executed in parallel.

The *concurrency* semantics can be used to specify whether concurrent accesses will take place and, if so, how the access pattern will look like. Depending on the level of concurrency, different algorithms might be appropriate for file system operations such as locking or metadata access. Concurrency semantics are performance-related by allowing simpler and faster centralized algorithms to be used when no concurrent access is happening. For instance, atomicity is only required for overlapping accesses.

The *consistency* semantics can be used to specify if and when clients will see modifications performed by other clients and applies to both metadata and data. This information can be used to enable client-side read caching whenever

[1] POSIX's synchronization functions `fsync` and `fdatasync` only allow synchronizing whole files even if this is not necessary.

possible. The consistency semantics is performance-related and can allow caching data and metadata locally.

The *ordering* semantics can be used to specify whether operations within a batch are allowed to be reordered. Because batches can potentially contain a large number of operations, the additional information can be exploited to optimize their execution. The ordering semantics is performance-related as it allows operations to be reordered for more efficient access. It is especially important to group operations of the same type to reduce the amount of network overhead.

The *persistency* semantics can be used to specify if and when data and metadata must be written to persistent storage. This can be used to enable client-side write caching whenever possible. The persistency semantics is performance-related and allows caching modified data and metadata locally. This can be especially advantageous when different levels of storage such as node-local SSDs are available as it allows writing the temporary data to the fast local storage without communicating via the network at all.

The *safety* semantics can be used to specify how safely data and metadata should be handled. It provides guarantees about the state of the data and metadata after the execution of a batch has finished. The safety semantics is performance-related by allowing to adjust the overhead incurred by data safety measures and to optimize network utilization by not waiting for unnecessary replies.

3 Related Work

The current HPC I/O stack has already been identified as problematic regarding future demands due to its complex layering and static architecture [3]. Even though there are a few approaches to provide configurable behavior and semantics in parallel distributed file systems, they are usually limited to single aspects of the file system or too static because they do not allow changes at runtime [17]. JULEA aims to solve these problems using its novel approach.

MosaStore is a versatile storage system that is configurable at application deployment time and thus allows application-specific optimizations [1]. This is similar to JULEA's approach. However, MosaStore provides a storage system bound to specific applications instead of a globally shared one. Additionally, the storage system can not be reconfigured at runtime and keeps the traditional POSIX I/O interface.

CAPFS introduces a new content-addressable file store that allows users to define data consistency semantics at runtime [22]. While providing a client-side plug-in API allows users to implement their own consistency policies, CAPFS is limited to tuning the consistency of file data and keeps the traditional POSIX interface. Additionally, the consistency semantics can only be changed on a per-file basis. JULEA covers a wider range of semantics and features a more fine-grained as well as a more dynamic approach.

Memory ordering and consistency are important factors in parallel programming for shared memory architectures, both for performance and correctness.

CPUs usually reorder memory load and store operations to improve performance [7,8]. Modern concepts such as those supported by C++11 and C11 allow developers to specify different constraints to achieve optimal performance while still maintaining correct execution of their applications [10]. JULEA's ordering semantics provide the same benefits by allowing the developer to provide additional semantical information to optimize execution.

ADIOS offers a novel and developer-friendly I/O interface that allows specifying the I/O configuration in an XML file that can be changed without recompiling the application [11,15]. Version 1.4 of ADIOS has added support for scheduling read operations. Several read operations can be scheduled using the `adios_schedule_read` function and then executed using the `adios_perform_reads` function. Read scheduling is very similar to JULEA's batches as it allows aggregating multiple operations for improved performance. However, JULEA's batches can contain arbitrary operations, making them more versatile.

4 Performance Evaluation

Benchmarks will be used to evaluate different performance aspects of JULEA and Lustre, which strives to support POSIX semantics. In addition to comparing JULEA to the other parallel distributed file system, a number of different semantics will be evaluated. However, due to the sheer amount of different semantics combinations, only those expected to have a significant impact on performance will be analyzed in more detail. JULEA's data performance will be evaluated using different atomicity, concurrency and safety semantics. A prior comparison of the metadata performance has been published in [12].

All evaluations have been conducted on the cluster of the Scientific Computing research group at the University of Hamburg. The benchmarks have been performed using a total of 20 nodes, with 10 nodes running the file system clients and 10 nodes hosting the file system servers. The nodes' hardware and software setup is as follows:

The client nodes each have two Intel Xeon Westmere EP HC X5650 CPUs (2.66 GHz, 12 cores total), 12 GB DDR3/PC1333 ECC RAM, a 250 GB SATA2 Seagate Barracuda 7200.12 HDD and two Intel 82574L Gbit Ethernet NICs. They run Ubuntu 12.04.3 LTS with Linux 3.8.0-33-generic and Lustre 2.5.0 (client); the MPI implementation is provided by OpenMPI 1.6.5.

The server nodes each have one Intel Xeon Sandy Bridge E-1275 CPUs (3.4 GHz, 4 cores total), 16 GB DDR3/PC1333 ECC RAM, three 2 TB SATA2 Western Digital WD20EARS HDDs, one 160 GB SATA2 Intel 320 SSD and two Intel 82579LM/82574L Gbit Ethernet NICs. They run CentOS 6.5 with Linux 2.6.32-358.18.1.el6_lustre.x86_64 and Lustre 2.5.0 (server).

To allow a proper assessment of the results, the following theoretical performance considerations should be kept in mind: The theoretical maximum performance of Gbit Ethernet is 125 MB/s. However, it is usually not possible to reach more than 117 MB/s due to overhead. Consequently, the maximum achievable performance between the clients and servers is approximately 1,170 MB/s.

The average round-trip time between the client and server nodes is 0.228 ms. Ignoring actual processing times, it is therefore possible to send and receive 4,386 requests/s.

The file systems' data performance will be evaluated using a large number of concurrently accessing clients that first write data and then read it back again; the write and read phases are completely separated and barriers ensure that only one type of operation takes place at any given time. To force the clients to read the data from the data servers during the read phase, the clients' cache was dropped after the write phase. The benchmark uses MPI to start multiple processes accessing the file systems in a coordinated fashion. There are two basic modes of operation: Individual files (each process accesses its own file, the individual files are accessed serially) and shared file (all processes access a single shared file concurrently in an interleaved fashion). All accesses use a variable block size and are non-overlapping, that is, no write conflicts occur. The file systems have been set up to provide ten data servers and one metadata server. Each benchmark has been repeated at least three times to calculate the arithmetic mean as well as the standard deviation.

4.1 Lustre

Lustre has been set up using its default options except for the stripe count that has been set to `-1` to enable striping over all available object storage targets (OSTs); the stripe size has been set to 1 MiB. While each OST has been provided by one of the servers' HDDs, the meta data target (MDT) has been provided by one of the SSDs. Lustre has been mounted using the client module as a normal POSIX file system with the `flock` option that enables support for file locking. The option should not have any influence on the benchmark results because they do not use file locking.

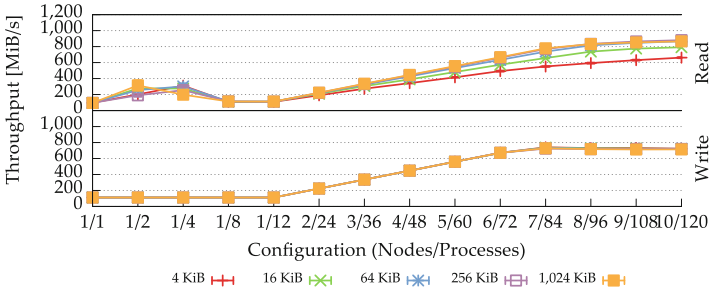


Fig. 3. Lustre: individual files via POSIX

Individual Files. Figure 3 shows Lustre's read and write performance when using individual files via the POSIX interface. Regarding read performance, it is interesting to note that configurations with a single node exhibit different

performance characteristics depending on the number of processes. While the configurations with one, eight and twelve processes all achieve a throughput of roughly 100 MiB/s, the configurations with two and four processes deliver 200–300 MiB/s; while this effect has to be related to some data being read from the cache of the operating system, the exact reasons for this are unclear. As explained earlier, the benchmark drops all caches between the read and write phases, therefore, this effect should not occur. The remaining configurations gradually deliver more performance as more nodes are added until reaching their maximum performance with ten nodes. As expected, smaller block sizes result in lower read performance due to additional overhead. However, it is interesting to note that even with a single process and a block size of 4 KiB, Lustre achieves a read performance of roughly 100 MiB/s. As mentioned previously, the Gbit Ethernet network can transfer at most 4,386 requests/s. Taking this into account, Lustre should only be able to read at a maximum of 17 MiB/s. This discrepancy is due to Lustre performing client-side readahead to increase performance. When considering write performance, it can be seen that all block sizes deliver the same performance. This is most probably due to Lustre's use of client-side write caching. Because individual files are used and each file is only accessed by one node, Lustre can utilize caching without sacrificing POSIX compliance.

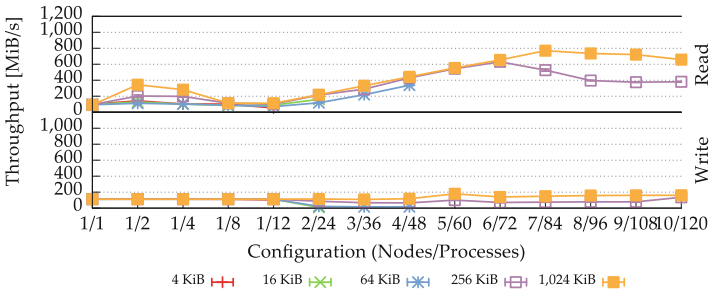


Fig. 4. Lustre: shared file via POSIX

Shared File. Figure 4 shows Lustre's read and write performance when using a single shared file via the POSIX interface. The read performance for the configurations using one node behaves in a similar way to the test case with individual files. For small block sizes, not all results could be collected because Lustre's performance was too low and the jobs exceeded the job scheduler's time limit. For 256 KiB and 1,024 KiB, the performance increases until six and seven nodes, respectively, and afterwards drops with each additional node. This result is surprising because only read operations are performed by all accessing clients, that is, no locking should be required. However, it appears that Lustre still introduces some overhead for these accesses, decreasing overall performance significantly. For the write phase, an interesting effect occurs: While using only a single node, performance is stable for all block sizes. When using more than one accessing

nodes, performance drops for all block sizes less than 1,024 KiB. As soon as multiple nodes are involved, Lustre has to send all write operations directly to the data server to achieve POSIX compliance. An additional factor for the low performance could be write locking that needs to be performed due to the concurrently accessing clients.

4.2 JULEA

JULEA has been configured to use a POSIX storage backend on the data servers' system HDDs. Additionally, JULEA was set to use a maximum of six client connections per node because it was observed that the default of twelve caused severe performance problems due to the large amount of TCP connections.

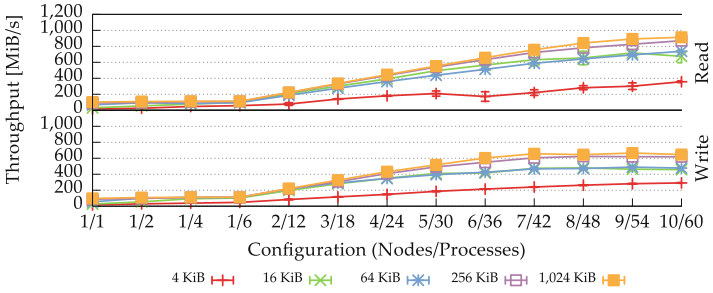


Fig. 5. JULEA: individual items

Default Semantics. Figure 5 shows JULEA's read and write performance when using individual items via the native JULEA interface. Regarding read performance, performance almost scales linearly until seven to eight nodes are used. Afterwards, the speedup slows down, reaching a maximum of more than 900 MiB/s using a block size of 1,024 KiB. As expected, smaller block sizes provide a lower overall performance with the exception of 16 KiB and 64 KiB that are reversed. Regarding write performance, the same effects as in the read case can be observed. Even though the performance does not increase with more than seven clients, it remains at a stable level.

Figure 6 shows JULEA's read and write performance when using a shared item via the native JULEA interface. During the read phase, the performance curve looks almost identical to its counterpart using individual items. While the performance speedup slowed slightly when going from nine to ten nodes using individual items, the shared item case is not affected by this drop. Additionally, the block size of 16 KiB provides a more stable performance curve. During the write phase, the performance curve looks less smooth than when using individual items. For instance, using the largest block size of 1,024 KiB, performance drops when increasing the number of nodes from five to six, only to rise again when using seven nodes. The fact that overall performance is lower than when using

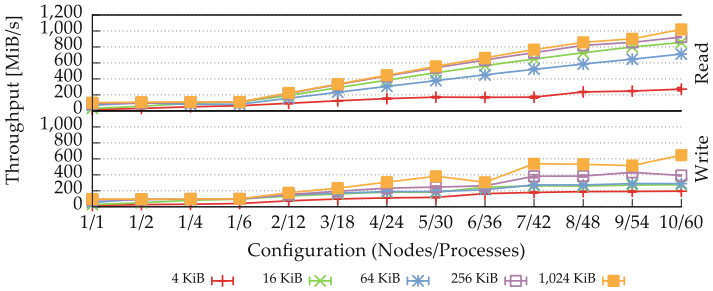


Fig. 6. JULEA: shared item

individual items indicates that the handling of shared files is suboptimal in the Linux kernel. Additional measurements using OrangeFS, different underlying file systems and JULEA's NULL storage backend have shown that these performance inconsistencies are not specific to JULEA, independent of the underlying file system and only occur if the file system is actually accessed using shared files.

To reduce the number of results and exclude the influences of the performance inconsistencies when using a single shared file, the following measurements have only been performed using individual items.

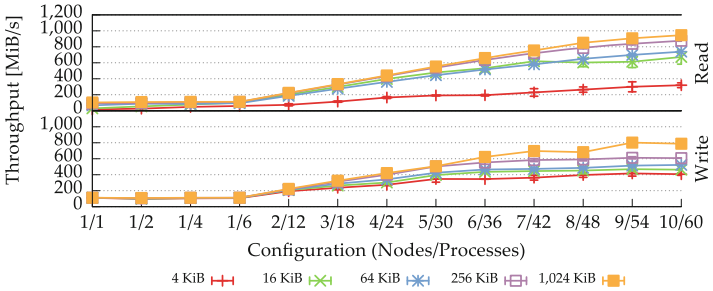


Fig. 7. JULEA: individual items using unsafe safety semantics

Safety Semantics. The following measurements have used the safety semantics to disable write acknowledgments for all write operations.

Figure 7 shows JULEA's read and write performance when using individual items via the native JULEA interface. During the read phase, there are only minor differences in performance in comparison to the default semantics. This is to be expected because the read operations are not handled differently depending on the safety semantics. During the write phase, performance is improved across the board for all block sizes. It is especially interesting to note that even a single process achieves the maximum performance of 110 MiB/s using a block size of

4 KiB because the clients do not have to wait for the write acknowledgments from the data servers. Using a block size of 4 KiB, the maximum performance is increased by 33 % when using ten nodes. The largest block size of 1,024 KiB manages to achieve a maximum performance of approximately 800 MiB/s, an improvement of 23 % when compared to the default semantics.

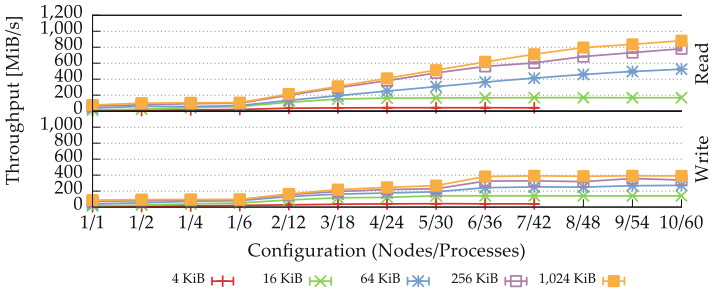


Fig. 8. JULEA: individual items using per-operation atomicity semantics

Atomicity Semantics. The following measurements have used the atomicity semantics to enforce atomic access for each read and write operation. JULEA currently implements atomicity using a centralized per-block locking algorithm.

Figure 8 shows JULEA's read and write performance when using individual items via the native JULEA interface. Regarding read performance, it is interesting to note that different block sizes show different scaling behavior: While the block sizes of 4 KiB and 16 KiB quickly reach a maximum and stay at this level, the remaining block sizes deliver more performance as more nodes are used. This behavior can be explained using a rough performance estimation: MongoDB manages to deliver roughly 20,000 inserts/s and 6,000 removes/s. Taking into account that each read or write operation requires one insert and one remove operation, a maximum of 13,000 operations/s can be performed.[2] This implies a maximum performance of roughly 50 MiB/s for a block size of 4 KiB and 200 MiB/s for a block size of 16 KiB. According to the measurements, 42 MiB/s and 170 MiB/s are reached for block sizes of 4 KiB and 16 KiB, respectively. Because a block size of 64 KiB can already support up to 800 MiB/s according to this approximation, the remaining block sizes' performance scales with the number of nodes. Interestingly, the largest block size of 1,024 KiB almost reaches the same performance as when using the default semantics. For smaller block sizes, the slowdown is more severe, however, resulting in a decrease of almost 30 %. Regarding write performance, the small block sizes manage to deliver almost the same performance as during the read phase. While the block size of 4 KiB reaches a maximum of 40 MiB/s, the block size of 16 KiB is limited to 140 MiB/s. The remaining block sizes perform much worse, however. This is due to the lower write performance that is already present when using the default semantics.

[2] This number is only intended to provide a rough estimate. In practice, the number might be lower due to the high discrepancy between insert and remove performance.

4.3 Discussion

The results demonstrate that the current state of parallel distributed file systems is mixed and that performance can be very hard to predict and understand. Even simple access patterns as the ones used for the presented benchmarks do not achieve the maximum performance. This is true for all tested file systems but has different reasons for each of them.

Lustre deals well with a large number of concurrent clients. This is most likely because Lustre can easily use the operating system's file system cache due to being implemented in kernel space. This allows Lustre to aggregate accesses and thus reduce the load on the servers. However, Lustre's performance is abysmal when accessing a single shared file as commonly done in scientific applications: Read performance decreases with more than seven client nodes and write performance does not scale beyond one client node. Consequently, only individual files are efficiently usable because it is not possible to inform Lustre about the application's I/O requirements to mitigate these performance problems.

JULEA's overall performance is held back by problems found within the underlying operating system and file systems. However, its dynamically adaptable semantics allow it to cater to a wide range of I/O requirements:

- Its default semantics enable performance results similar to those of Lustre when using large block sizes. Lustre has advantages for small block sizes due to its client-side caching and readahead functionalities. However, these advantages vanish as soon as shared files are used.
- The safety semantics can be used to reduce the network overhead by not awaiting the data servers' replies. This is similar to Lustre's default behavior when using individual files.
- Atomic operations can be achieved by using the atomicity semantics. While the performance of large read operations is not reduced significantly, write operations suffer a performance penalty of up to 40 %. However, using JULEA's fine-grained semantics, it is possible to use atomic operations only when absolutely necessary.

In contrast to Lustre, JULEA can be adapted to different applications by setting its semantics appropriately. While it is neither possible to improve Lustre's shared file performance due to its POSIX compliance nor to use other file systems such as OrangeFS for workloads requiring overlapping writes, it is possible for JULEA to support and to be tuned for these specific use cases.

5 Conclusion and Future Work

This paper presents a new approach for handling application-specific I/O requirements in HPC. The JULEA framework includes a prototypical implementation of a parallel distributed file system and provides a novel I/O interface featuring dynamically adaptable semantics. It allows applications to specify their I/O requirements using a fine-grained set of semantics. Additionally, batches enable the efficient execution of file system operations.

The results obtained in this paper demonstrate that there is need for I/O interfaces that can adapt to the requirements of applications in order to provide adequate performance for a variety of different use cases. The current circumstances effectively force application developers to adapt their applications to work around limitations found in specific file systems in order to achieve the best possible performance. An indication for this is the wide variety of I/O libraries, such as SIONlib [6], that deal with particular file system constraints. This can significantly increase the development and maintenance overhead because applications have to be optimized for different file systems' semantics instead of being able to optimize the file systems according to their I/O requirements.

The concept introduced by the JULEA framework fills the gap by allowing applications to adapt the file system to their exact I/O requirements instead of the other way around. The available results show that the supplementary semantical information can be used to adapt the file system's behavior in such a way as to optimize performance for specific use cases. Additional results and more in-depth information about JULEA are available in [13].

Even though JULEA provides a convenient testbed to experiment with different semantics and prototype new functionality, it is necessary to provide dynamically adaptable semantics for established I/O interfaces and parallel distributed file systems for widespread adoption of these new features. These interfaces have to be standardized and supported by a sufficiently large subset of file systems to provide consistent functionality across different implementations.

First of all, it is necessary to agree on default semantics suited for modern HPC applications and a common set of parameters that should be configurable. The semantics presented in this paper are meant to provide a good starting point for further evaluation.

5.1 Future Work

As mentioned previously, it is often unreasonable to port applications to new I/O interfaces due to their size and complexity. Thus, to avoid having to rewrite applications to be able to make use of JULEA's novel features, some form of compatibility would be preferable. Because many applications already use high-level I/O libraries such as ADIOS or NetCDF, JULEA could be integrated into applications by providing backends for these I/O libraries. As ADIOS's API design is relatively close to JULEA, a thin backend would be sufficient to enable all ADIOS-aware applications to use JULEA without any further modifications. ADIOS makes use of XML-based configuration files to specify the applications' I/O, which could be easily extended to add more semantical information about the actual data, similar to what has been done in [14].

References

1. Al-Kiswany, S., Gharaibeh, A., Ripeanu, M.: The case for a versatile storage system. SIGOPS Oper. Syst. Rev. **44**(1), 10–14 (2010)

2. Bartz, C.: An in-depth analysis of parallel high level I/O interfaces using HDF5 and NetCDF-4. Master's thesis, University of Hamburg, April 2014
3. Brinkmann, A., Cortes, T., Falter, H., Kunkel, J., Narasimhamurthy, S.: E10 – Exascale IO, May 2014. http://www.eiow.org/home/E10-Architecture.pdf?attredirects=0&d=1. Accessed: April 2015
4. Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: a parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, pp. 317–327. USENIX Association
5. Cluster File Systems Inc.: Lustre: a scalable, high-performance file system, November 2002. http://www.cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf. Accessed: November 2014
6. Frings, W., Wolf, F., Petkov, V.: Scalable massively parallel I/O to task-local files. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009. ACM, New York (2009). http://doi.acm.org/10.1145/1654059.1654077
7. Gharachorloo, K., Gupta, A., Hennessy, J.: Performance evaluation of memory consistency models for shared-memory multiprocessors. In: Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV, pp. 245–257. ACM, New York (1991). http://doi.acm.org/10.1145/106972.106997
8. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory consistency and event ordering in scalable shared-memory multiprocessors. In: Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA 1990, pp. 15–26. ACM, New York (1990). http://doi.acm.org/10.1145/325164.325102
9. The IEEE and The Open Group: Standard for Information Technology - Portable Operating System Interface (POSIX) Base Specifications, Issue 7. IEEE Std 1003.1, 2013 Edition (incorporates IEEE Std 1003.1-2008, and IEEE Std 1003.1-2008/Cor 1–2013) pp. 1–3906, April 2013
10. ISO/IEC JTC 1/SC 22 - Programming languages, their environments and system software interfaces: ISO/IEC 9899:2011 - Information technology - Programming languages - C, December 2011
11. Klasky, S., Liu, Q., Lofstead, J., Podhorszki, N., Abbasi, H., Chang, C., Cummings, J., Dinakar, D., Docan, C., Ethier, S., Grout, R., Kordenbrock, T., Lin, Z., Ma, X., Oldfield, R., Parashar, M., Romosan, A., Samatova, N., Schwan, K., Shoshani, A., Tian, Y., Wolf, M., Yu, W., Zhang, F., Zheng, F.: ADIOS: powering I/O to extreme scale computing. In: SciDAC 2010 Conference Proceedings, pp. 342–347 (2010). http://computing.ornl.gov/workshops/scidac2010/papers/data_q_liu.pdf
12. Kuhn, M.: A semantics-aware I/O interface for high performance computing. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2013. LNCS, vol. 7905, pp. 408–421. Springer, Heidelberg (2013)
13. Kuhn, M.: Dynamically adaptable I/O semantics for high performance computing. Ph.D. thesis, University of Hamburg, Germany, November 2014 (to be published)
14. Kunkel, J., Minartz, T., Kuhn, M., Ludwig, T.: Towards an energy-aware scientific I/O interface - stretching the ADIOS interface to foster performance analysis and energy awareness. Comput. Sci. Res. Dev. **27**(4), 337–345 (2011)
15. 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 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE 2008, pp. 15–24. ACM, New York (2008)

16. Message Passing Interface Forum: MPI: a message-passing interface standard. Version 3.0, September 2012. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. Accessed: November 2014
17. Patil, S., Gibson, G.A., Ganger, G.R., Lopez, J., Polte, M., Tantisiroj, W., Xiao, L.: In search of an API for scalable file systems: under the table or above it? In: Proceedings of the 2009 Conference on Hot Topics in Cloud Computing, HotCloud 2009. USENIX Association, Berkeley (2009)
18. Rew, R., Davis, G.: Data management: NetCDF: an interface for scientific data access. IEEE Comput. Graph. Appl. **10**(4), 76–82 (1990). http://dx.doi.org/10.1109/38.56302
19. The HDF Group: Hierarchical data format version 5, July 2014. http://www.hdfgroup.org/HDF5. Accessed: November 2014
20. The TOP500 Editors: TOP500, June 2014. http://www.top500.org/. Accessed: November 2014
21. Vilayannur, M., Lang, S., Ross, R., Klundt, R., Ward, L.: Extending the POSIX I/O interface: a parallel file system perspective. Technical report ANL/MCS-TM-302, October 2008. http://www.mcs.anl.gov/uploads/cels/papers/TM-302-FINAL.pdf
22. Vilayannur, M., Nath, P., Sivasubramaniam, A.: Providing tunable consistency for a parallel file store. In: Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies, FAST 2005, vol. 4. USENIX Association, Berkeley (2005)
23. Wikipedia: Festplattenlaufwerk - Geschwindigkeit, November 2014. http://de.wikipedia.org/wiki/Festplattenlaufwerk#Geschwindigkeit. Accessed: November 2014
24. Wikipedia: Mark Kryder - Kryder's Law, November 2014. http://en.wikipedia.org/wiki/Mark_Kryder#Kryder.27s_Law. Accessed: November 2014