Scalable Software Distribution for HPC-Systems Using MPI-Based File Systems in User Space

Jakob Dieterle
GWDG
Göttingen, Germany
e-mail: jakob.dieterle@gwdg.de

Hendrik Nolte
GWDG
Göttingen, Germany
e-mail: hendrik.nolte@gwdg.de

Julian Kunkel
Department of Computer Science
Georg-August-Universität Göttingen
Göttingen, Germany
e-mail: julian.kunkel@gwdg.de

***Abstract*—Even on state-of-the-art high-performance computing systems, data-intensive tasks that are running on a lot of nodes can encounter waiting times when accessing large files. This can be caused by a bottleneck in the network bandwidth. To this end, this paper aims to develop a filesystem that utilizes inter-node communication to more efficiently distribute large files between nodes and reduce the bandwidth usage on the network. The presented file system was implemented using the Filesystem in Userspace interface and the Message Passing Interface. During evaluation, it showed promising performance compared to a slower native file system but did not reach the performance level of an optimized file system. A refined version was able to outperform the slower native file system with 16 nodes by 8%, achieving a bandwidth-reduction to latency-increase ratio of approximately 22.**

Keywords-data distribution; optimized reading; one-sided communication.

I. Introduction

With the steady increase of demand for computing power for use cases both in research and industry and the slowdown of Moore's law [1], high-performance computing (HPC) systems have to increase their number of nodes to keep up. This growing number of nodes imposes significant challenges on the I/O performance of large-scale HPC systems [2]. Distributing data, container files, or software packages to hundreds or thousands of nodes for a single job can lead to long waiting times before any processing can even begin [3]. The bottleneck in such situations is the bandwidth of the storage nodes hosting the relevant files.

The comparable low available bandwidth originates from different sources. For instance, large HPC systems are increasingly built by partitioning the overarching system into different compute islands [4]. A high-speed interconnect connects all nodes within each island, while the connections between different compute islands have a comparatively low bandwidth. Since the overarching software stack does not necessarily change drastically between the different islands, it is preferable from an administrative perspective to centrally manage and export a global software stack to the individual islands. Therefore, accessing software typically requires going through an interconnect with a limited bandwidth.

To tackle this issue, we want to develop a mechanism that reduces the bandwidth usage on the interconnect to the storage nodes by utilizing inter-node communication. Therefore, access through the remote software stack is only done once, and software distribution to potentially hundreds of nodes is done via the local, high-speed interconnect. The challenge is to find an efficient way to distribute the data between the nodes while reading the file's content only once. The best way to implement such a mechanism in a user-friendly way is to include it in a custom file system. Using the Filesystem in Userspace (FUSE) interface will allow us to develop and run the file system in user space, which is necessary with standard user privileges. We will implement the file system using the Message Passing Interface (MPI) to facilitate inter-node communication since it is the most commonly used interface for message passing. Our file system is supposed to run on the computing nodes, handling the access of files provided by an existing distributed file system. Popular examples of distributed file systems are *BeeGFS*, *Lustre*, *GPFS*.

The main contributions of this work are:

- Presenting multiple designs for a file system to reduce overall bandwidth usage to the storage nodes
- Implementation of two design approaches using different communication paradigms
- Benchmarking the implemented file systems with various configurations

The remainder of the paper is organized as follows: Section II presents the used technologies and related work. In Section III, we outline the design of the file system. Section IV is about evaluating the implemented file system, including methodology, results, and discussion. Finally, Section V contains the conclusion, and we discuss future work.

II. Background and Related work

To make file systems available in user space, the FUSE kernel module was incorporated into the Linux kernel with version 2.6.14 [5], which consists of the kernel module and the libfuse userspace library. By linking the libfuse library to a program, a non-privileged user can mount their own file system by writing their own open/read/write, etc. methods. This allows the implementation of custom file systems that do not necessarily require a dedicated storage device but can instead use forward storage requests to an underlying file system. For example, Fuse-archive by Google [6] allows the user to mount different archive file types (.tar, .zip, etc.) and access it like a regular directory while decompressing the data on the fly.

Rajgarhia and Gehan evaluated the performance of FUSE using the Java bindings as an example [7]. They found that for

block sequential output, FUSE adds a lot of overhead when dealing with small files and a lot of metadata but becomes quite efficient with larger files. When running the PostMark benchmark, FUSE added less than 10% compared to native ext3.

Vangoor et al. also analyzed the performance of FUSE and its kernel module design in greater detail [8]. According to Vangoor et al., FUSE can perform with only 5% performance loss in ideal circumstances, but specific workloads can result in 83% performance loss. Additionally, a 31% increase in CPU utilization was measured.

The most commonly used standard for passing messages between nodes is the *Message Passing Interface* (MPI). MPI provides different concepts for communication, such as point-to-point communication (send, receive), collective communication (broadcast, gather, allgather), and one-sided communication (get, put).

The performance of MPI can vary depending on the environment and implementation that is being used. Hjelm analyzed the performance of one-sided communication in OpenMPI [9]. The paper provides an overview of OpenMPI's RMA implementation and evaluates its performance by benchmarking the `Put`, `Get`, and `MPI_Fetch_and_op` methods for latency and bandwidth. The benchmarks showed constant latency for `Put` and `Get` for messages of up to 2^{10} bytes and a drastic latency increase for messages larger than 2^{15} bytes. Analog to the latency results, the bandwidth performance plateaus with message sizes larger than 2^{15} bytes.

There are also alternatives to MPI for message passing and I/O management. One is Adios2, presented by Godoy et al. [10]. Adios2 is designed to be an adaptable framework to manage I/O on various scales, from laptops to supercomputers. Adios2 provides multiple APIs with its MPI-based low-level API designed for HPC applications. It realizes both parallel file I/O and parallel intra/interprocess data staging. Adios2 adopts the Open Systems Interconnection (OSI) standard and is designed for high modularity.

With the increasing complexity of HPC applications, the complexity and size of software packages used for these tasks also increase. Zakaria et al. showed that package dependencies in HPC have increased dramatically in recent years [11]. Different well-known software deployment models are discussed in this paper, including store models like spack, which is used on Sofja. The authors also present their solution, Shrinkwarp, which reduces loading times for highly dynamic applications. The paper focuses more on software distribution models and package management than improving loading times by increasing I/O efficiency.

Creating file systems in user space to improve I/O performance is not new. There are already several other file systems with similar goals. For example, FusionFS [12]. FusionFS is a file system in user space that supports intensive metadata operation by storing metadata for remote files locally and is optimized for file writes.

The concept of disaggregation in HPC systems aims to decouple resources like memory and processing power by allowing direct memory access over network interfaces. Peng et al. conducted a study on memory utilization, showing that 90% of all jobs utilized less than 15% of node memory and 90% of the time less than 35% of node memory is used [13].

Above, we presented existing projects that aim to improve I/O performance. However, none of those focus on enhancing performance for distributing large files from a few storage nodes to many compute nodes.

III. Design

Before designing a mechanism to distribute files with our file system, a decision has to be made on which communication paradigms MPI offers should be used. The obvious answer to that might be collective communication since, with the broadcast operation, distributing data from one node to all other nodes is very easy and efficient. However, using collective communication also imposes very difficult to overcome limitations. In case there are one or more nodes in the job that don't access a file that all the other nodes need to access, all other nodes would get stuck when trying to broadcast the data, or all nodes would have to be forced to join the broadcast, even if they don't need to access the broadcasted data. But even if all nodes want to access the same file, they would also have to access the blocks of the file in the same exact order, which is not something we can expect. Point-to-point communication can also be very efficient. However, it always needs interaction from both involved nodes, which means distributing data to a lot of nodes would have to be organized in a predetermined way. The complexity of such a mechanism would most likely scale very severely with a large number of nodes. MPI's One-Sided communication methods are known to be less efficient but allow nodes to access designated parts of the memory of the other nodes by utilizing remote memory access. With the `MPI_Get` method, we can read data from other nodes without interrupting the process on the target node. Using One-Sided communication also gives us more flexibility since we don't have to synchronize the processes between nodes, simplifying the mechanism and reducing busy waiting times. For these reasons, we decided to design a mechanism specifically using the `MPI_Get` method.

To fully use the benefits of the `MPI_Get` method during file access, we want to ensure the entire file is already available to be accessed only using direct memory access between the nodes. To that end, the file is split into N parts, with N being the number of nodes. In the `open` method of our FUSE file system, each node reads its assigned part of the file from the storage nodes. This process is predetermined for all nodes so each node can create a lookup table to know which node holds each part of the file (see Listing 1).

When a node calls the read method of our file system to access a part of the file, it will check which node or nodes are holding the needed data. If the data is already in the local buffer the data can just be returned from the buffer, otherwise, the data has to be obtained from one or more nodes using the `MPI_Get` method (see Listing 2). If data was retrieved from other nodes, it would be written to the `file_buffer`, and

```
def my_open(path):
    file_handler = open(path)

    file_buffer = MPI_Win_allocate
        (file_handler.size, char)
    meta_buffer = int[file_handler.size]

    meta_buffer = calculate_distribution
        (file_handler.size, world_size)

    offset, size = calculate_my_range
        (file_handler.size, my_rank)
    data = read(offset, size)
    file_buffer[offset:offet+size] = data

    return file_handler
```

Figure 1. Pseudo code for open method

```
def my_read(file_handler, offset, size):

    if (in_buffer(offset, size)):
        return file_buffer[offset:offset+size]

    targets = get_targets(offset, size)
    for target in targets:
        t_offset, t_size = caclulate_target_range
            (meta_buffer, offset, size)
        data = MPI_Get(t_offset, t_size, target)
        file_buffer[t_offset:t_offset+t_size] = data

    meta_buffer[offset:offset+size] = my_rank

    return file_buffer[offset:offset+size]
```

Figure 2. Pseudo code for read method

the `meta_buffer` will also be updated accordingly so that we don't have to retrieve the data multiple times.

With this design approach, the bandwidth usage to the storage nodes is minimized to a workload of just $1 \cdot filesize$, instead of $N \cdot filesize$, since each part of the file is only read once during the open method. Afterward, the nodes only communicate with each other to distribute the data. In the following sections, we will refer to this design as the One-Sided-Reading (OSR) file system. Additionally to this design approach, a simple design using a broadcast operation in the read method was implemented to compare performance between the two communication paradigms. We will refer to this implementation as Naive design approach or simple broadcast approach.

IV. Methodology

This work's main focus is read timings, which are important variables in terms of the scalability of our file systems. To evaluate the implemented file system's performance, tests will be conducted over a range of nodes and file sizes.

The tests were conducted on a smaller tier 3 HPC system. That means all nodes are shared by default and not exclusively allocated for each job. The *Sofja* System consists of 30 Nodes, with each node providing two AMD EPYC 7763 64-core processors on two sockets, totaling 128 cores per node and 3840 cores for the whole system. Each node also provides 512 GB of memory. The cluster is split up into two racks and uses HDR100-InfiniBand fabric. The nodes have access to different storage systems. The StorNext file system is used to access the 3 PiB of Home directories. For intense I/O applications, the dedicated Scratch file system can be used. The Scratch file system offers more than 500 TiB of storage space, from which more than 100 TiB are NVMe drives. The Scratch file system provides much better I/O performance and higher bandwidth than the Home directories and runs on BeeGFS. The 30 nodes that will be used for testing also provide local SSDs that offer temporary storage per job and memory-based storage on `/dev/shm`.

Six use cases will be tested by accessing files with `sha256sum`: access over the native file system, access over our One-Sided Reading file system (OSR), copying the file to local SSDs with the Naive approach before access (Bcast to local), copying the file to `/dev/shm` with the Naive approach before access (Bcast to /dev/shm), accessing the file directly with the Naive approach (Bcast no copy, possible, since `sha256sum` accesses the file sequentially like `cp`) and access over the simple FUSE Passthrough. These six use cases will also be tested with the Home directory as the original source of the file and the Scratch file system as the source. That results in twelve total test scenarios.

Each scenario will be tested on all combinations from the number of nodes and file sizes. With the number of nodes being: 1, 2, 4, 8, 16, and 24. And file sizes of 1KB, 100KB, 10MB, and 1GB. This means all ten scenarios will be tested with 24 configurations. Each configuration will be tested ten times to receive statistically robust result data. Although the cluster offers 30 nodes, we only tested with up to 24 nodes since not all nodes are available at all times.

To guarantee the resulting timings are valid, we have to ensure the accessed files can not be cached on the nodes between tests since we want to measure the time it takes to actually read the file from the storage nodes over the network. The best way to guarantee this would be to drop the page caches between runs. However, that requires sudoer rights, which we didn't have. Thus the random test data was generated for each individual test, that way the file's content changes between each test and thus cannot be accessed from caches. However, just writing the file to the desired location can cause the file to be in the page caches of the node that is generating and writing the file. To ensure this doesn't affect the measurements, we will run the benchmarker with one extra node, which only generates the file but doesn't run any tests.

All tests are organized into runs. Each run tests one combination of the number of nodes and file size for all six use cases on either the Home file system or Scratch file system. For each test run, three jobs are needed: the One-Sided Reading file system, the Naive file system, and the Python benchmarker, which executes test file generation, all tests, and writes the resulting data into a `csv` file. When testing on n nodes, the benchmarker has to be run with $n + 1$ nodes so that the first node of the benchmarker job is not included in the n nodes of

the jobs for the two file systems. The benchmarker generates the test files using `/dev/urandom`. The time needed to calculate the hash sum is measured using MPI's `MPI_Wtime` method.

V. Results

The results of the described benchmarks can be seen in Figure 3, with more details listed in Table I. First, we look at the results when accessing the files over the Home directory. With small files, you can see that the native file system is the fastest, and the number of nodes doesn't affect the performance much. With the 10 MB file, the native file system starts to lose performance with a steady increase of time needed with more than four nodes. Between one node (0.1 sec.) and 24 nodes (0.23 sec.), the native file system is 2.3 times slower. The effect is more pronounced with the 1 GB file, where it is 2.91 times slower (from 7.35 sec. to 21.41 sec.). During testing with the 1GB file and 24 nodes, it achieved a throughput of 1.121 GB/s. The three different broadcast use cases also have no visible loss in performance with the 1KB file but start to lose performance beginning with the 100KB file and more than four nodes, where all three perform very similarly. The Naive design performs similarly with the 10 MB file when copying to local SSDs or to `/dev/shm`. When copying to SSDs, the procedure is 2.85 times slower with 24 nodes (0.4 sec.) compared to 1 node (0.14 sec.). When copying to `/dev/shm`, it is 3.58 times slower (from 0.12 sec. to 0.43 sec.). When accessing the file directly, the Naive design starts to be much slower, with more than eight nodes. That takes 5.69 times longer when going from 1 node (0.13 sec.) to 24 nodes (0.74 sec.). The trend is continued for the 1 GB file. From the three broadcast use cases, copying the to `/dev/shm` achieves the highest throughput on 24 nodes with the 1 GB file of 0.689 GB/s, while accessing the file directly only achieves 0.331 GB/s with the same configuration. The One-Sided Reading design is the worst file system with small files and more than two nodes. It is also the only file system that gets significantly slower with an increasing number of nodes on the 1KB file. With the 10 MB file, the scaling of this file system is already better than any of the broadcast use cases. Here, it is 2.56 times slower when going from 1 node (0.16 sec.) to 24 nodes (0.41 sec.). With increasing file size the loss in performance decreases, with the 1 GB file, it is only 1.62 times slower (from 14.28 sec. to 23.09 sec.). When accessing the 1 GB file with 24 nodes, the One-Sided Reading file system is much faster than the three broadcast use cases and only 1.68 sec. or 7.85 % slower than the native file system, achieving a throughput of 1.039 GB/s. In general, the results for the Passthrough file system are very similar to the performance of the Home file system.

When we look at the results when using the `/dbnscratch` file system, we have slightly different results for our implemented file systems. With just one node, all file systems are faster than when using `home`; with 24 nodes, all broadcast use cases and the One-Sided Reading file system are slower compared to using `home`. That is also reflected in increased multipliers when comparing the performance with one node against 24 nodes, which can be found in Table I. The Scratch

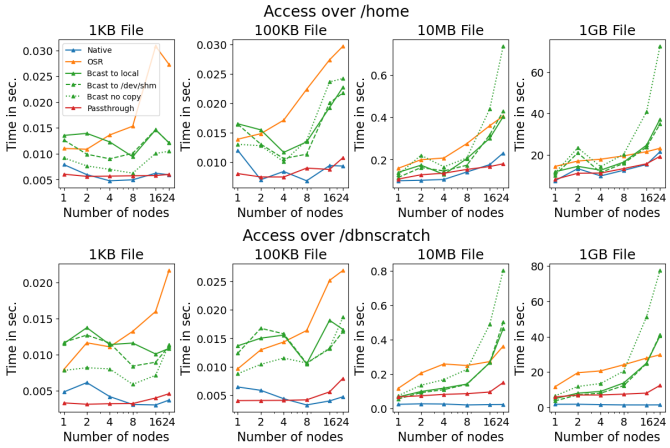

Figure 3. Time measurements on Sofja system

TABLE I. Result comparison for tests with 1 GB file

	Time with 1 node (in seconds)	Time with 24 nodes (in seconds)	Increase	Throughput with 24 nodes
/home	7.355	21.408	2.910	1.121 GB/s
OSR	14.289	23.094	1.617	1.039 GB/s
Bcast to local	11.703	37.161	3.175	0.646 GB/s
Bcast to /dev/shm	9.991	34.826	3.486	0.689 GB/s
Bcast no copy	11.124	72.505	6.518	0.331 GB/s
Passthrough	8.021	19.192	2.393	1.251 GB/s
/dbnscratch	1.772	1.427	0.806	16.819 GB/s
OSR	11.607	29.737	2.562	0.807 GB/s
Bcast to local	4.923	40.404	8.207	0.594 GB/s
Bcast to /dev/shm	3.625	41.221	11.372	0.582 GB/s
Bcast no copy	6.478	77.450	11.956	0.310 GB/s
Passthrough	5.872	12.426	2.116	1.931 GB/s

file system itself however is much faster than the Home directory file system, even with a 1 GB file size it even shows a performance increase when going from 1 to 24 nodes, achieving a throughput of 16.819 GB/s. The simple Passthrough file system shows similar performance in the beginning. The better performance with small files and a few nodes can also be attributed to the deviations between nodes again. Starting with the 10 MB file, a significant difference can be observed compared to the Scratch file system for any number of nodes.

It would be interesting to know how much each factor actually affects performance. Comparing the overhead caused by MPI and FUSE would be important to identify where the file system can be improved. To this end, another test was conducted with the OSR file system and the Naive file system. With the OSR file system, the `xmp_open` method and the `xmp_read` method were timed. In the `xmp_read` method of the Naive file system, the time it takes to read the requested range of bytes was measured, as well as the time it takes to broadcast the read data. For both file systems, a counter was added to count how many times the `xmp_read` method was called on each node. Using this information we can calculate the time the operations of the two file systems take in total. For the OSR file system, we take the average of the `xmp_open`

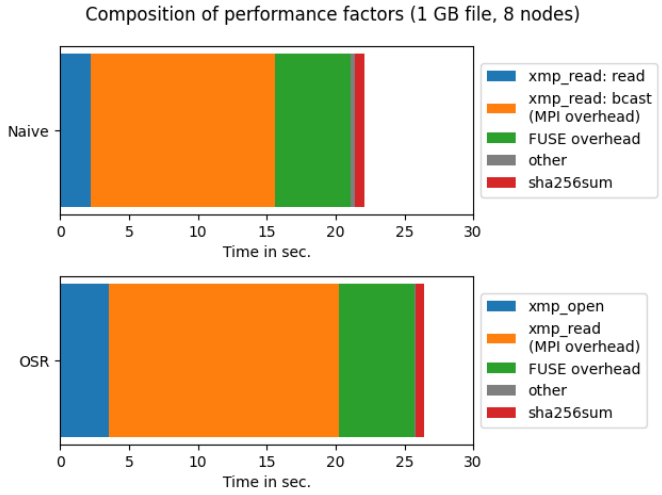

Figure 4. Composition of performance factors for the OSR and Naive file system

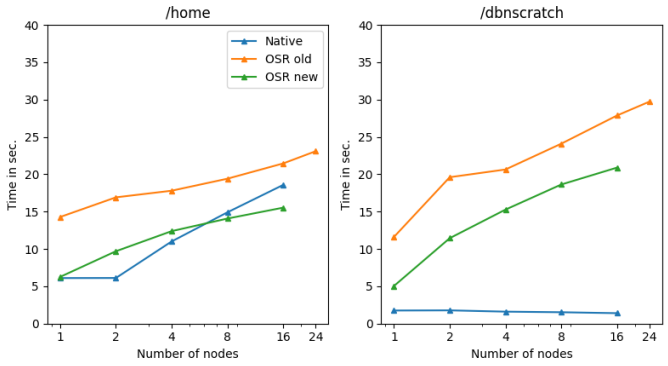

Figure 5. Timings of OSR file system without metadata buffer compared to with metadata buffer

timings over the nodes to get the value for this method. For the `xmp_read` method, we take the average over all calls of this method on all nodes and then multiply this value by the number of times the method was called on each node. The same is done for the two timings we take in the `xmp_read` method of the Naive file system. We can use the test results visualized in Figure 3 to quantify the overhead FUSE introduces. We can subtract the result for the Scratch file system from the same result for the Passthrough file system when accessing the file over the Scratch file system. This difference can be attributed to the FUSE overhead since the simple Passthrough file system does not add any features. It only passes the file system calls through the FUSE kernel module. A more detailed view of how the FUSE overhead is calculated and what it consists of can be seen in the following equation.

$$\begin{aligned}\text{FUSE-Overhead} &= time(\text{FUSE Passthrough}) - time(\text{native}) \\ &= time(\text{FUSE context switches} \\ &\quad + \text{ executing FUSE method} \\ &\quad + \text{ native FS call}) \\ &\quad - time(\text{native FS call})\end{aligned} \tag{1}$$

Together with the result of measuring the time `sha256sum` takes to calculate the hash-sum when the file is already in the memory (0.653 seconds), we can deduct those measurements from the total time it took to calculate the hash-sum during this test, and we receive a remaining time, that should be very small and can be attributed to the fact that we combine different test results and other smaller factors that we were not able to measure. The test was conducted by accessing a 1 GB-sized file on the Scratch file system with eight nodes, as most of the system's nodes were occupied by other long-running jobs.

The total time the test took was similar to before: 21.432 seconds for the Naive file system and 26.426 seconds for the OSR file system. Both results are a little bit higher than in the previous tests; this could be caused by the job mentioned that was running concurrently. The `xmp_open` method of the OSR file system took 3.5 seconds on average over the nodes. The `xmp_read` calls took around 0.002 seconds on average over all nodes with the method to be called 7630 times, resulting in a total time for the `xmp_read` method of 16.7112 seconds. The time we calculated for the FUSE overhead is 5.557 seconds. When subtracting these results and the 0.653 seconds for calculating the hash-sum from the total time, the remaining time of 0.004 seconds can be attributed to other factors. The `xmp_read` method of the Naive file system was also called 7630 times. The average time for reading the range of bytes was 0.00028, totaling 2.214 seconds. The broadcast operation took an average of 0.00175 seconds, totaling 13.355 seconds. Again, the time we calculated for the FUSE overhead is 5.557 seconds. Subtracting those values and the 0.653 seconds for the hash-sum from the total time, the remaining 0.305 seconds can be attributed to other factors. These results are visualized in Figure 4.

Unfortunately, only after these tests did we realize that a lot of performance was wasted by the very expansive metadata operations needed for the simple caching mechanism. Before, the location for every byte was stored in the metadata with the according node ID. The metadata buffer was used to determine on which node the needed data was located or if it was already stored locally. Such a high resolution on the meta resulted in vast amounts of required operations, especially when working with large files. To test how much performance was lost, another small set of tests was run without using a metadata buffer at all. Instead, the nodes where the requested data is located were inferred using the same algorithm to split the file and populate the metadata buffer in the `xmp_open` method. The results of these tests, compared to the old results, can be found in Figure 5. We can see that we save at least 5 seconds without the meta buffer operations. This means this version of the OSR file system is more efficient than the native home file system starting with only eight nodes.

VI. Discussion

On the Home and Scratch file systems with small file sizes, the custom file systems are slower than the native file systems. In the case of the broadcast-based file systems, this is caused by the MPI overhead, while for the OSR file system, this is caused by the `xmp_open` method and the MPI overhead. Compared to the One-Sided Reading design, the broadcast-based file systems don't show any performance loss with an increasing number of nodes here since the file can probably be broadcasted with only one operation. In contrast, the One-Sided Reading design must start transferring data between nodes in smaller batches. With increasing file sizes, the broadcast-based file systems get significantly worse with increasing numbers of nodes. The limiting factor here might be that the nodes must wait a significant amount of time between broadcast operations until all nodes have reached the next broadcast operation. That gets worse with more nodes and adds up with larger files, requiring a more significant number of broadcast operations. It is unexpected, though, that accessing the file directly with the Naive file system is so much slower with large files than copying the file first. That might be caused by extra waiting time when waiting for the next broadcast when some nodes need more time to process the received data inside the `sha256sum` algorithm than other nodes. The native Home file system starts to show its limitations with the 10 MB file already and has a maximum throughput of slightly above 1 GB/s. While the One-Sided Reading file system is significantly worse than the Home file system with a file size of up to 10 MB, it also reaches a maximum throughput of around 1 GB/s with the 1 GB file (see Table I). But more importantly, the scaling from one node to 24 nodes for the OSR file system (1.617) is better than for the Home file system (2.910) when accessing the 1 GB file. The hypothesis is that with just some more nodes (32 or more), the OSR file system might start to be faster than `/home` if both trends continue with a more significant number of nodes. This is likely because the performance of the Home file system is expected to worsen with an increasing number of nodes because of the bandwidth bottleneck. In contrast, we can expect the performance of the OSR file system to continue to grow linearly as the bandwidth of the communication channels between the nodes is not limiting the performance at this scale. It is clear that the MPI overhead limits the Naive design approach and OSR file system. The simple Passthrough shows a similar performance to the Home file system. The FUSE overhead is not so noticeable here since the Home file system's latency is already high.

It is also interesting to note that we have some unexpectedly high results; for example, with 10 MB and two nodes and with 1 GB and two nodes, the timings for most file systems are unexpectedly higher even compared to four nodes. That might be due to 'noisy neighbors' also stressing the network and storage nodes. Curiously, the OSR file system does not seem to be affected by it as much. That might be because it only reads the test file during the open method and relies on communication between the nodes afterward, thus not affected as much by 'noisy neighbors'. However, that is only speculation and is difficult to verify.

When looking at the results when accessing the file over the Scratch file system, the results are similar for the FUSE file systems. The Scratch file system itself proves to be much faster than Home. That can be due to various aspects, such as better network bandwidth over a fabric connection, multiple storage nodes sharing the load, better storage devices, etc. It is also notable that for all file sizes, the performance gets better with an increasing amount of nodes, which can probably be attributed to caching on the storage servers themselves. The results of the Passthrough file system show the overhead caused by FUSE, especially with the 1 GB file. The Passthrough file system is multiple seconds slower than Scratch, caused by the multiple context switches, as mentioned in Section II. This performance loss also contributes to the results of the other custom file systems. We also do not have unexpected spikes, as we observed when using the Home file system, since 'noisy neighbors' have less effect on our jobs when the native file system does not reach its bandwidth limit as easily.

The small remaining time for both file systems confirms that our measurements are accurate and that we identified the most critical performance factors. In the case of the Naive file system, the MPI overhead is associated with the time the broadcast operation takes in the `xmp_read` method (orange tile). For the OSR file system, the MPI overhead is associated with the entire time `xmp_read` method takes (orange tile) since the file is already entirely in memory, distributed over the nodes, and all operations in the `xmp_read` method are related to remotely accessing the file using MPI. The FUSE overhead is the same for both file systems since the FUSE overhead is dependent on the number of file system operations, which is not influenced by the file system itself. The FUSE overhead should also be the same for any number of nodes since the number of file system calls per node is unaffected by the number of nodes running the job. For both file systems, the FUSE overhead is significantly less than the time the MPI commutation takes, while the overhead caused by the MPI communication will increase with a growing number of nodes. Since this work aims to develop a file system that scales well over an increasing number of nodes, we can confidently say that the MPI communication overhead is the most significant factor for performance. Limiting the MPI overhead should be the first priority to improve the performance of the file system as a whole. This could be done by reducing the number of times the `MPI_Get` method is called to a minimum, for example, by always reading the entire chunk of the file that is assigned to a node so that each node never has to communicate with another node more than once. This would increase the number of times that a read can be covered by the local buffer and decrease the number of times the `MPI_Get` method is called.

After analyzing the results of the performance factor analysis, we noticed that a significant amount of time spent during the `xmp_read` method of the OSR file system is caused by a large number of metadata operations interacting with the metadata

buffer. This was mainly due to the very high resolution of the metadata buffer, keeping track of the location of each byte. To quantify the resulting performance impact, an updated version of the file system was implemented without any metadata buffer, eliminating the metadata operations. Removing the buffers means we don't have any caching mechanism, which imposes a performance loss in most use cases. However, we wanted to test the performance without the caching mechanism since we suspected it to be extremely inefficient and impact the results more than it should. The resulting performance improvement is significant, with a 5-second improvement for all test configurations. When comparing the new results to the performance of the native file systems, the updated version exceeds the performance of the Home file system with only eight nodes. This performance gain is expected to grow with an increasing amount of nodes. The optimized Scratch file system is still much faster. However, we also did not create a workload that caused a bottleneck for this file system. It is important to note that the performance difference from the first version of the OSR file system depends on the size of the accessed file, not on the number of nodes accessing the file.

VII. Conclusion and Future Work

In conclusion, an MPI-based FUSE file system is presented that can use two different methodologies to orchestrate I/O from multiple nodes. For highly synchronous reading operations, a broadcast mechanism that could read a 1 GB file using less than 4% of the bandwidth compared to the baseline is shown. This entailed a performance penalty of approximately 250% during the synthetic benchmark. In real-world scenarios, this performance penalty might decrease if the available bandwidth is also shared with other users. Therefore, one achieves a bandwidth-reduction to latency-increase ratio of approximately 8.

The more flexible OSR mechanism reduced the performance penalty to around 8% while retaining the same bandwidth reduction, resulting in a bandwidth reduction to latency increase ratio of approximately 22. The refined version of the OSR file system improved performance even further. It was able to reduce latency compared to the baseline while also retaining the bandwidth reduction.

Future work should focus on testing the OSR file system on a larger scale. Some technical improvements should also be made, like developing an efficient caching mechanism, running the file system multi-threaded on each node, and adding the capability to handle multiple opened files simultaneously.

References

[1] C. E. Leiserson *et al.*, "There's plenty of room at the top: What will drive computer performance after moore's law?", *Science*, vol. 368, no. 6495, eaam9744, 2020. DOI: 10.1126/science.aam9744.

[2] S. Lang *et al.*, "I/o performance challenges at leadership scale", in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09, Portland, Oregon: Association for Computing Machinery, 2009, pp. 1–12, ISBN: 9781605587448. DOI: 10.1145/1654059.1654100.

[3] W. Frings *et al.*, "Massively parallel loading", in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13, Eugene, Oregon, USA: Association for Computing Machinery, 2013, pp. 389–398, ISBN: 9781450321303. DOI: 10.1145/2464996.2465020.

[4] M. A. Mollah *et al.*, "A comparative study of topology design approaches for hpc interconnects", in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 392–401. DOI: 10.1109/CCGRID.2018.00066.

[5] T. kernel development community, "The linux kernel documentation - fuse", [Online]. Available: https://www.kernel.org/doc/html/next/filesystems/fuse.html?highlight=fuse (visited on 10/24/2024).

[6] Google, "Fuse-archive repository", [Online]. Available: https://github.com/google/fuse-archive (visited on 10/24/2024).

[7] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems", in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 206–213, ISBN: 9781605586397. DOI: 10.1145/1774088.1774130.

[8] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of User-Space file systems", in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, Santa Clara, CA: USENIX Association, Feb. 2017, pp. 59–72, ISBN: 978-1-931971-36-2.

[9] N. Hjelm, "An evaluation of the one-sided performance in open mpi", in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16, Edinburgh, United Kingdom: Association for Computing Machinery, 2016, pp. 184–187, ISBN: 9781450342346. DOI: 10.1145/2966884.2966890.

[10] W. F. Godoy *et al.*, "Adios 2: The adaptable input output system. a framework for high-performance data management", *SoftwareX*, vol. 12, p. 100 561, 2020, ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561.

[11] F. Zakaria, T. R. W. Scogland, T. Gamblin, and C. Maltzahn, "Mapping out the hpc dependency chaos", in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2022, pp. 1–12. DOI: 10.1109/SC41404.2022.00039.

[12] D. Zhao *et al.*, "Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems", in *2014 IEEE International Conference on Big Data (Big Data)*, 2014, pp. 61–70. DOI: 10.1109/BigData.2014.7004214.

[13] I. Peng, R. Pearce, and M. Gokhale, "On the memory underutilization: Exploring disaggregated memory on hpc systems", in *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2020, pp. 183–190. DOI: 10.1109/SBAC-PAD49847.2020.00034.