Scalable Software Distribution for HPC-Systems with Software Pools Using MPI and 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—**Despite the increasing computing power of high-performance computing (HPC) systems, complex tasks on large-scale clusters can still be hindered by significant waiting times when loading large software packages and dependencies. These delays are often caused by network bandwidth bottlenecks, which can severely impact application performance. To address this challenge, this paper presents a new way of distributing software in HPC systems. Our software pools can hold whole software stacks in a single file, while our implemented tools can distribute software pools efficiently to large clusters while reducing bandwidth usage to a minimum. Software pools offer additional advantages, such as portability, reproducibility, and security, while seamlessly integrating into existing environments using Lmod.**

Keywords-file distribution; optimized reading; containerization.

I. INTRODUCTION

This paper extends our previous work on scalable software distribution for high performance computing systems with MPI-based file systems in user space, which introduced a design for a file system with the main goal to reduce bandwidth usage when reading large files [1]. While the original paper showed promising results, the presented file system had some limitations that were caused by the general design approach to tackle the given problem. Thus, this paper presents the new concept of software pools, instead of iterating on the work from the previous paper. Software pools are designed to improve performance while distributing software in high performance computing (HPC) environments, by reducing complex software stacks into compact image files. This allows the usage of collective communication, instead of one-sided communication used in the previous paper, potentially improving the performance and reducing complexity. This paper also provides new benchmarking results, which are compared to the results of the previous paper.

The challenge of distributing large files in HPC environments is becoming more important, as the increase in demand for computing power in data centers is unbroken. This is especially true, with the recent surge of artificial intelligence and the popularization of large language models. Because of the slowdown of Moore's law, HPC systems have to keep up with the demand by increasing the total number of cores and nodes inside the systems [2]. However, the increasing level of parallelization also leads to a greater demand for networking bandwidth inside HPC systems. 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], [4].

We believe that the distribution of large files to many nodes could be organized more efficiently by using tools such as the Message Passing Interface (MPI) and Filesystem in Userspace (FUSE) [5]. In our previous paper [1], we presented a design for a file system in user space that uses MPI to distribute data between nodes on demand while the nodes access the file. This version of our file system used MPI's `MPI_Get` method to directly access memory on other nodes, and was thus called the One-Sided-Reading (OSR) file system. While this approach reduced bandwidth usage to a minimum and showed promising scaling in our performance test, it had some technical limitations, and the performance was not competitive against optimized file systems. The biggest limitation of the previous design was the on-demand communication approach. The overhead introduced by FUSE and MPI were the biggest performance factors, and both are related to the number of total read calls to the file system. The amount of individual read operations on a file is only influenced by the size of the blocks with which the application is accessing the file. We cannot control the block size the application is working with, so the room for improvement is relatively limited. The implementation of the file system also lacked some basic functionality that would be needed for real-world applications. Including those features would have increased complexity and possibly reduced performance as well.

In this paper, we want to investigate a different approach to the problem. Instead of distributing data on demand, files will be efficiently distributed before the user application accesses them. We will present our concept of software pools, which are image files containing whole directory trees which can be easily distributed and mounted anywhere. Multiple tools are implemented to build, distribute, and mount software pools. Those tools are benchmarked to evaluate their performance and viability for real-world applications and to compare this approach to the design introduced in our previous paper.

The main contributions of this work are:

- Presenting the concept of software pools
- Implementing tools that enable the utilization of software pools in HPC environments
- Benchmarking these tools and comparing them to previous designs

```
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 remainder of the paper is organized as follows: Section II presents related work and used technology. Section III presents the concept of software pools and related tools. Section IV outlines the methods used for for benchmarking the implemented tools. The results of these benchmarks are presented in Section V. In Section VI, we discuss the presented results. Finally, Section VII contains the conclusion, and we discuss future work.

II. Background

In our previous paper, we presented a file system in user space, which uses MPI to reduce bandwidth usage when accessing large files in the context of HPC systems [1]. This design used MPI's one-sided communication methods to transfer data between nodes on demand. Upon opening a file, it is split up between the nodes, and each node loads its associated part of the file into memory, which is available to be used with one-sided communication (see Figure 1). When accessing a certain part of the file, the file system checks whether it is already in the node's memory. If that is not the case, the node will read the missing range of bytes from the node that initially loaded that part of the file using the `MPI_Get` method (see Figure 2).

In our performance tests, the file system was able to match or even outperform a slower existing file system providing the

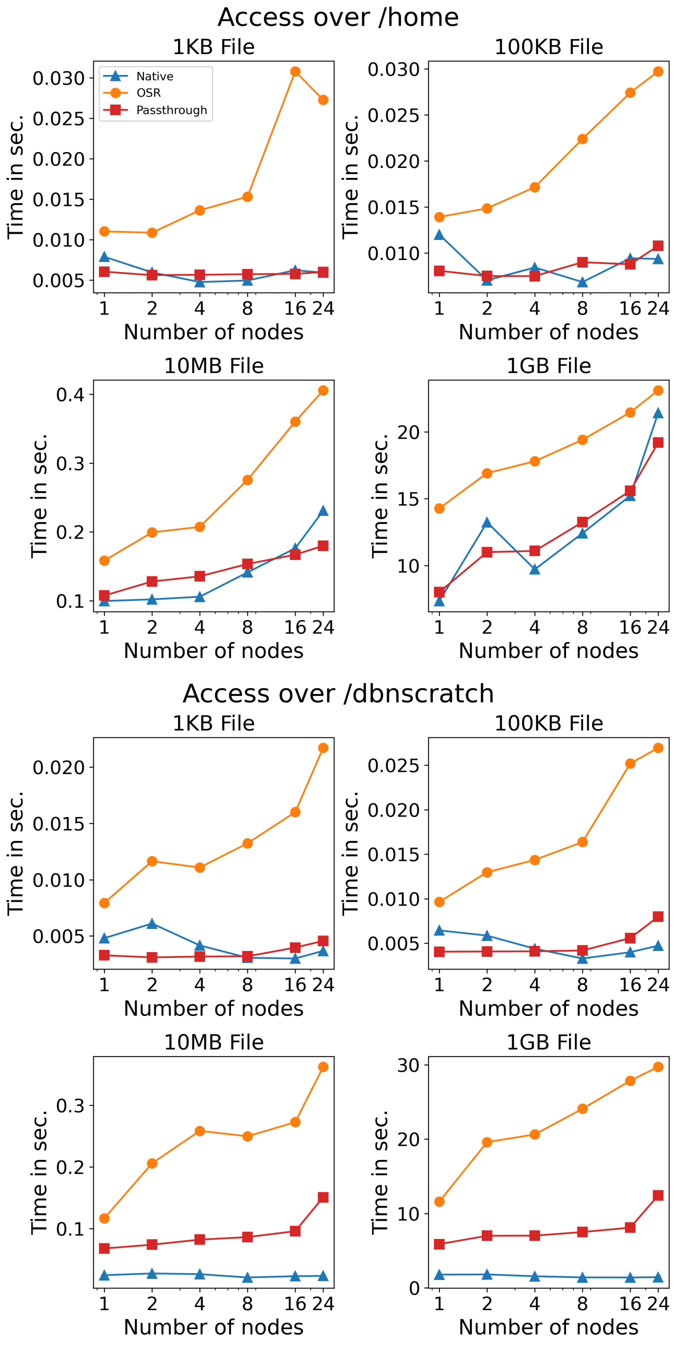

Figure 3. Time measurements on Sofja system.

home directory with about 1.4 Gb/s bandwidth. However, it was not able to reach the performance level of a more optimized scratch file system. At least not with the workloads we tested during our benchmarks (see Figure 3).

We conducted further tests with more granular timing measurements to analyze what factors were most impacting the performance of our file system. The goal was to isolate the impact MPI and FUSE have, specifically. The results show that the MPI communication calls are the biggest factor affecting the performance of the file system (see Figure 4). Since the

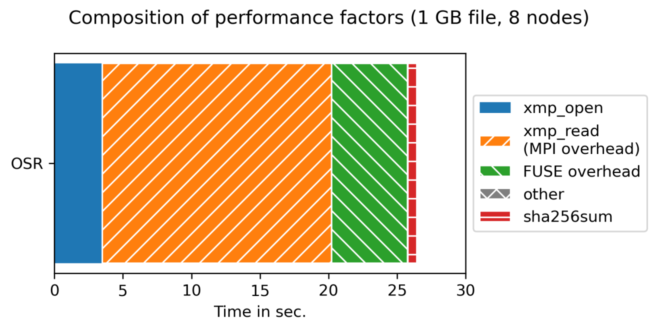

Figure 4. Visualization of performance factors for the OSR file system.

overhead by FUSE is only dependent on the file size and block size while accessing the file, it is constant for varying numbers of nodes. At the same time, the amount of overhead introduced by MPI will increase with a larger number of nodes.

Originally, file systems in Linux exclusively operated on the kernel level. Thus, developing new file systems requires changing and compiling the kernel, which can be very cumbersome and is only possible for privileged users. To make developing new file systems more accessible, the FUSE (Filesystem in Userspace) kernel module was incorporated into the Linux kernel with version 2.6.14 [5]. The FUSE project 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. When the library issues a syscall, the VFS in the kernel handles it and passes it on to the FUSE kernel module. The FUSE kernel module then calls the linked user space program, the FUSE file system, which finally executes the call.

FUSE is mainly used to implement virtual file systems, which don't actually store data but provide a different view or translation of an existing file system or storage device. Other accessible resources can also be used by FUSE file systems. For example, Fuse-archive by Google [6] allows the user to mount different types of archive file types (.tar, .zip, etc) and access it like a normal directory while decompressing the data on the fly. It uses buffers to increase performance and prevent decompressing the same files multiple times. The file system implemented later for this work will be a virtual file system using FUSE. Some popular distributed file systems also use FUSE, such as *GlusterFS* [7].

The most commonly used standard for passing messages between nodes is the *Message Passing Interface* (MPI). MPI provides different concepts for communication like Point-to-Point communication, Collective communication, and One-Sided communication. Point-to-point communication involves two specific processes to send a message from one to the other. It requires both processes to call respective methods, such as `MPI_Send` and `MPI_Recv`. Developers need to ensure that these methods are actually called by both processes. Otherwise, one process may get stuck while waiting for the other involved process, possibly resulting in a deadlock. There are also non-blocking alternatives allowing asynchronous Point-to-Point communication (`MPI_Isend` and `MPI_Irecv`). Collective communication always involves a group (or groups) of processes to share data. For example, processes can send a message to all other processes of the specified group using `MPI_Bcast` or gather data from all other processes with `MPI_Gather`, while `MPI_Allgather` combines both operations into one. These examples also have non-blocking alternatives. For synchronization when using point-to-point or collective communication, the `MPI_Barrier` method is most commonly used. As the name suggests, it acts as a barrier for the specified group of ranks, at which all ranks have to wait until all ranks reach this point in the program. One-sided communication uses the concept of Remote Memory Access (RMA), which allows a process to share data with another process without interrupting it. During initialization, all processes need to specify a memory 'window' that will be accessible by other processes. The method `MPI_Win_allocate` creates the window and allocates its memory. For the actual communication, the following two methods are of main interest. `MPI_Get` reads a range of bytes from the window on a specified target, `MPI_Put` writes into the memory of a specified target, and `MPI_Accumulate` realizes a reduction operation over the same memory over multiple targets. When using any of these RMA communication methods, they need to be encapsulated by synchronization methods, mainly `MPI_Win_lock` and `MPI_Win_unlock`. These methods start and terminate a RMA communication epoch, in which accessing the specified window is possible. `MPI_Win_lock` takes a specifier as an argument with the options `MPI_LOCK_SHARED` and `MPI_LOCK_EXCLUSIVE`. When using the keyword `MPI_LOCK_EXCLUSIVE` the window is completely locked for any other operations on the window. Other `MPI_Win_lock` calls trying to access the window on the same rank have to wait until the current lock is released again by `MPI_Win_unlock`. The keyword should be used when writing operations are executed on the window. Using `MPI_LOCK_SHARED` is safe as long as only read operations are issued during the RMA epoch, and it allows multiple processes to start an epoch on the same window at the same time. In this situation, the usage of `MPI_Win_lock` is still necessary, as the method also has to initialize the communication between the two ranks since RMA is not entirely one-sided in MPI. MPI provides many more methods and concepts, the ones presented here are a selection that will be relevant for the rest of this work.

III. Related Work

FUSE is said to significantly affect the performance of file operations due to the additional context switches introduced between user space and kernel space, as described above. Rajgarhia and Gehan evaluated the performance of FUSE using the Java bindings as an example [8]. 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% in comparison to native ext3. Vangoor et al. also analyzed the performance of FUSE and its kernel module design in greater detail [9]. According to Vangoor et al., FUSE can perform with only 5% performance loss in ideal circumstances, but certain workloads can result in 83% performance loss. Additionally, a 31% increase in CPU utilization was measured. This negative effect on performance will be relevant when we compare our file system's performance to the native file system later.

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 [10]. OpenMPI has supported one-sided communication since version 1.8.0 but emulated it using point-to-point communication. With version 2.0.0, OpenMPI introduced an implementation of one-sided communication using actual RMA concepts. 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 basically constant latency for `Put` and `Get` for messages of up to 2^{10} bytes and a drastic increase of latency 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 alternative APIs to MPI for message passing and I/O management. One of them is Adios2, presented by Godoy et al. [11]. Adios2 is designed as an adaptable framework for managing I/O on a wide range of scales, from laptops to supercomputers. Adios2 provides multiple APIs with its MPI-based low-level API being 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. Each provided component can be mapped to OSI layers 4 to 7. At the core of its concept are abstract engines, which describe I/O workflows by bundling components from OSI layers 4 to 7. Engines are highly adaptable to different use cases (mainly parallel file I/O and parallel data staging) and performance needs.

Also important to mention here is OpenMP (Open Multi-Processing). OpenMP provides libraries for multi-threaded processing using a shared memory model. In C/C++ `#pragma`'s are used to start multi-threaded processing, they are often used to parallelize loops without data dependency, thus multiple iterations of a loop can be calculated concurrently. OpenMP is limited to multi-threaded processing on a single node, so we cannot use it for the current project. However, MPI and OpenMP are often used together with MPI handling inter-node communication and OpenMP used for parallelization inside the nodes.

As High-Performance Computing (HPC) applications become increasingly complex, the size and complexity of the software packages used to support them have also grown. Research has shown that the number of package dependencies in HPC has skyrocketed in recent years [4]. This paper, written by Zakira et al., explores various software deployment models, including store models like Spack, which is used on the HPC systems hosted by the GWDG. The authors also introduce their own solution, Shrinkwarp, which aims to reduce loading times for highly dynamic applications. However, the paper's primary focus is on software distribution models and package management, rather than optimizing loading times through improved I/O efficiency.

The concept of creating file systems in user space to enhance I/O performance is not a novel idea. Several existing file systems, such as FusionFS, have already explored this approach. FusionFS, in particular, is a user-space file system that optimizes metadata operations by storing remote file metadata locally and is designed to handle high-volume file writes [12]. In contrast, in our work, we focus on optimizing file reads to facilitate the distribution of files and software.

In HPC Systems, resources like memory and CPUs are tightly coupled into nodes. Systems may offer different configurations of nodes to serve different use cases. Still, studies show, that HPC jobs often underutilise the available memory [13]. 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 the node less than 35% of node memory is used [13]. These results highlight the under utilization of resources by a majority of jobs running on HPC Systems, while only few jobs profit from well equipped compute nodes. A possible approach to improve resource utilization could be the concept of disaggregated architectures, as they aim to disconnect different resources like computing power and memory capacity from each other. Thus, the paper introduces different disaggregated architectures, including a centralized design with dedicated memory nodes and a decentralized design, in which nodes share their local memory. Finally, Peng et al. present their implementation of a remote paging library *rMap*. It uses a centralized approach to allow nodes with little local memory to run memory-intensive jobs by requesting memory pages of dedicated nodes and swapping them into local memory.

Creating and distributing software environments is a common problem, especially in the HPC environment. A popular solution for this problem is containerization. A container is a lightweight and portable way to package an application, its dependencies, and a runtime environment into a single unit that can be easily deployed and managed. Containers allow the execution of software in an environment totally isolated from the host system. Similar to virtual machines, while not requiring dedicated resources and using the kernel of the host operating system. The best containerization software in the HPC environment is Apptainer. Apptainer is designed with a focus on security and reproducibility. Rajesh Pasupuleti et. al. discussed what advantages Apptainer containers offer for running AI applications on HPC systems [14].

In HPC Systems, software packages are often managed by module systems. The HPC systems hosted by the GWDG use Lmod. Lmod is a Lua-based module system that uses module files to dynamically change the user's environment [15]. This paper aims to offer a solution for distributing software that seamlessly integrates into the existing Lmod environment on the GWDG systems.

In this section we reviewed different frameworks for communication including OpenMP, Adios2 and MPI. OpenMP does not fit our given problem, as it only allows communication between processes on the same node, and not between multiple nodes. Adios2 allows for communication between nodes, but mainly provides more high level API which is unnecessarily complex for our use case. We also talked about other file systems that aim to improve file I/O performance like FusionFS. However, in contrast to our goal of improving performance when reading large files, FusionFS is focused on improving performance for writing to files. Containerization applications like Docker and Apptainer were also mentioned. They allow running software in isolated systems, similarly to virtual machines, but with less overhead by using the same kernel as the host system. We considered using container files as a base for our software pools, but decided against it, because container images often include all the files of a operating system and can be difficult to handle.

Ultimately, the goal of this paper is to create a tool set that includes features of many of the before mentioned technologies, such as file I/O performance improvement, image file handling and software distribution.

IV. Design

The proposed design of our original paper introduced a lot of overhead caused by the on-demand communication approach. Using MPI calls to transfer data while files are being accessed introduces a lot of extra latency, which is a problem for performance-critical applications. Especially in the HPC environment, where users might be charged per used core hour.

For this reason, we want to investigate whether it is more efficient to distribute the files before the application starts. Specifically, we want to use MPI to broadcast the files and store a copy locally so that the local copy can be accessed during runtime. However, this becomes quite complicated and inefficient when dealing with large directory trees and large amounts of files. For example, simple Python environments can easily contain tens of thousands of files. To simplify the task of broadcasting whole software stacks and huge directory trees, we want to pack them into a single file that is easy to distribute and mount on the target nodes. These files, which can hold complex software stacks, will be called software pools. Once the software pool is mounted on the target nodes, we want to integrate it automatically into an existing software module system. In our case, that would be Lmod.

Being able to load multiple software packages by mounting a single file has multiple other advantages. It reduces the Inode usage and metadata operations on the original file system. This is especially true when dealing with a lot of files, as is the case when using Python, Conda, or similar environments. Software pools can make it easier to manage collections of software packages on multiple levels. We plan to provide software pools on personal, project, and data center level, so that users can easily modify their environment for their current tasks. Software pools make sharing and reproducing software collections easy, as they can include binaries and source code. That allows the software pools to be shared between users and different HPC Systems. Software pools offer a simple way to add a trust factor for integrity and trustworthiness by offering a built-in method to sign and verify pool files using existing algorithms with private and public key pairs.

```
# create file pool.ext2 which contains the
# software pool
dd if=/dev/zero of=pool.ext2 bs=<block-size> \
    count=<number-blocks> conv=sparse

# create file system and copy given directory
# as root directory into the pool file
mke2fs pool.ext2 -d <source-directory>
```

Figure 5. Bash commands for building a software pool.

Choosing the correct file format for the project is crucial. The first idea was to use apptainer containers as software pools. Being able to quickly build containers from a simple definition file can be very beneficial for software pools' use cases. The directory tree of an apptainer container can be mounted by dumping the container's content using apptainer's `sif` tool. The resulting squashFS image could then be mounted using `squashfuse`. However, while testing this workflow, multiple drawbacks became obvious. Most importantly, apptainer images often take quite a long time to build and cannot be modified without completely rebuilding them. That makes the process of creating a software pool unnecessarily cumbersome and time-consuming.

Instead, we decided to focus on simpler file types and we used image files as containers for ext2, ext3, or ext4 file systems. To handle all operations in the context of the files representing software pools, we implemented our own tool called `sw-pool`. This tool offers the following commands: build, sign, verify, and load.

The build command creates a software pool from a given source directory. It takes the output file and source directory as arguments and the file size as an option. This directory should already include everything the pool should contain, most importantly, the binaries of the applications included in the pool. First, a file with a fixed size has to be created. This file should be sparse to ensure it only occupies as much disk space as needed. Then, the file can be written to using the `mke2fs` tool. It is part of the E2fsprogs package, which includes various utilities to handle ext2 (or ext3, ext4) file systems. With `mke2fs`, we can initialize the file system and copy the content of the software pool in one step, as you can see in Figure 5. This image can later be mounted using the `load` command.

The sign and verify commands add a factor of trustworthiness and integrity to the software pools. Internally, we use OpenSSL with the `-sign` and `-verify` to sign and verify the file with a given private and public key pair.

The load command broadcasts the given software pool, writes it to local temporary storage, and mounts it to the given mount point. The broadcasting and mounting procedures are done with other standalone tools we implemented for this use case. These tools are separated from the primary software pool tool, as broadcasting and mounting files with dedicated tools can also be very helpful outside of the context of software pools.

When the load command is called, it first determines the ideal temporary directory to store the local copy of the software pool. Then, our tool `scalable-fs-cp` is called. It takes an input file or directory and the target path as arguments. Before it does anything else, it checks if the input file already exists in all of the nodes it is running on. If the file is already present, nothing has to be done, so the tool can simply exit. Only if the file doesn't exist on all nodes does it start the procedure to broadcast the file using MPI.

Before the file can be broadcasted, it has to be read into memory. This is not done by one node, instead the file is split into N sections, with N being the number of nodes in our job. This is more efficient, as we know from our previous paper. After the sections of the file are loaded into memory, it can be broadcasted. For that, we iterate over the nodes, and each node broadcasts its section to the other nodes. This is not done with a single big chunk, however. Since our software pools are sparse files, the file size can be quite large, even though only a small part of the file was actually written. The rest of the file is filled with zeros, as we used `/dev/zero` as input when creating the file. To avoid broadcasting all the empty parts of the file, each section is processed block by block, with a default block size of 4096 bytes. Each will only be broadcasted if it contains a byte that is not zero. This should significantly reduce the time it takes to broadcast large pool files that are partly empty without having a big impact on broadcasting fully written pools, as we check for the first nonzero byte for each block. For this to work, the array to store the file in memory should be initialized to zeros. This can be done easily by simply using `calloc` instead of `malloc` in C.

Once the file is broadcasted, it can simply be written to a given target location on the nodes. If the tool detects that it is only running on a single node, it skips the broadcasting procedure and simply uses the `cp` command to copy the file to the given target location. Pseudo code for this procedure is listed in Figure 6. Copying directories is also supported. This comes with a performance loss, however, since the directory has to be compressed to and extracted from an archive file before and after broadcasting.

Once the distribution of the software pool is done, the software pool tool uses our `scalable-fs-mount` tool to mount the local copy to a given mount point. This tool supports different file types, in the case of our software pool file, it uses `fuse2fs`, which is also part of the E2fsprogs package.

Now that we have successfully distributed and mounted the software pool, the included software must be made available to the user. Theoretically, the user could just add the binary

```
def bcast_cp(input_file, output_path,
        rank, world_size):
  if file_exists:
    return 0
  if world_size > 1:
    file_buffer = zeros(file_size(input_file))
    offset, range = get_section(rank)
    file_buffer[offset:offset+range] =
            input_file[offset:offset+range]
    for curr_rank in world_size:
      blocks = split(file_buffer, curr_rank)
      if not is_empty(block):
        MPI_Bcast(block, curr_rank)
    write(output_path, file_buffer)
  else:
    copy(input_file, output_path)
```

Figure 6. Pseudo code for scalable-fs-cp.

```
<Pool-Name>
└─<Pool-Version>
   ├─modulefiles
   │  └─<Pool-Name>
   │     └─<Pool-Version>
   │        └─<SW-Name>
   │           └─<SW-Version>.modulefile
   ├─install
   │  └─<SW-Name>
   │     └─<SW-Version>
   │        └─binaries...
   ├─source
   │  └─<SW-Name>
   │     └─<SW-Version>
   │        └─source code...
   └─config
      └─...
```

Figure 7. Proposed Software Pool Structure.

locations manually to the path. While this might be feasible for small software pools, it would not work for more complex software stacks, since most HPC systems already use module systems that can solve this problem for us, especially. In our case, the HPC systems hosted by the GWDG use the Lmod module system. As described in Section II, Lmod uses module files to load modules into the user's environment. Lmod looks for module files based on the `MODULEPATH` environment variable. To make our software pool visible for Lmod, we need to provide the correct module files and add the corresponding paths to the `MODULEPATH` variable. This happens in the `gwdg-sw load` command after mounting the software pool. For this process to reliably work, the pools need to have a standardized format so that the `gwdg-sw` tool knows where to find the module files. Our proposed pool structure is presented in Figure 7.

The directory structure for the module files might seem unnecessarily complex. However, it is needed for Lmod to detect the correct module and software package hierarchy. This is complicated by the fact that Lmod requires the module files

```
local version = myModuleVersion()
local pkgName = myModuleName()
local mountPoint = os.getenv("GWDG_SW_MOUNT")
local poolName = "test-pool"
local poolVersion = "0.5"
local pkg = pathJoin(mountPoint,poolName,poolVersion
                ,"install",pkgName,version,"bin")
prepend_path("PATH",pkg)
```

Figure 8. Module File Example.

to be in a different directory than the module that the module file itself points to. The module file, in this case, would add the path of the binaries of the corresponding software in the `install` directory. An example for what a module file should look like can be seen in Figure 8.

The `source` and `config` directories are optional. However, pool authors should consider making the pools reproducible. To support this, any source code would go into the `source` directory, and any other files that are required for building the pool should go into the `config` directory.

With this in place, all the software packages included in the software pool are visible for Lmod after loading the pool with `gwdg-sw load <pool-name>/<pool-version>` and can be activated with Lmod using `module load <SW-name>/<SW-version>`.

V. Method

It is difficult to compare the performance of the tools we introduced in this paper to the file system we implemented in our previous paper, as they follow two completely different approaches. The file system from our previous paper impacted the latency of each read operation during the execution of the user's application. The software pool tools we implemented in this paper distribute the software packages before the user application starts. Thus, the effect on read latency is reduced because we still have overhead caused by FUSE, but no overhead from MPI. The overhead from MPI is shifted to the loading procedure which happens before the start of the user application. To see if this improves the overall situation, we can compare the total time it takes to read a file with the OSR file system with the time it takes to distribute the file in a software pool and read it afterward.

The tests will be run on the Emmy system hosted by the GWDG [16]. The system consists of 1.423 nodes with 111.464 cores. The system scored 5,95 PetaFlop/s during the LINPACK benchmark.

For comparison with the OSR file system, the results from the previous paper will be used. In this paper, we will focus our benchmarks on files with a size of 1 GB since the project aims to improve performance with large software stacks, and the largest test case from the previous paper is 1 GB.

For the first part of our benchmarks, we will measure the time it takes to distribute a pool using our introduced `sw-pool` tool with 2, 4, 8, 16, and 32 nodes. Ten runs will be conducted for each number of nodes, and the average of the ten runs will be calculated to obtain a robust result. To match the benchmarks of our previous paper, we will create a software pool, that holds a file with the size of 1 GB filled with random data. The commands to create our pool for tests are listed in Figure 9. Note that the file for the software pool has to be slightly larger than the test file since it needs some space to create the ext2 file system. To ensure that caching mechanisms do not affect our testing results, the test data will be regenerated on a different node before each run.

This software pool will be saved to the scratch file system. We will measure the overall time the command for mounting the software pool takes. The exact command we will use to execute and measure the test is listed in Figure 10. Here, the `$LOCAL_TMPDIR` variable points to the local SSDs of the nodes, and the `$SCRATCH` variable points to the SSDs of the remote scratch file system. In order to isolate how much of the total time is used for reading and broadcasting the file, additional time measurements are added to the code using the `MPI_Wtime` method.

```
# create test file
head -c 1GB /dev/urandom > test-pool/random.data

# create pool inlcuding test file
sw-pool build -s 1020MB test-pool.ext2 test-pool
```

Figure 9. Procedure to create software pool for testing.

```
time mpirun sw-pool -v load -m $LOCAL_TMPDIR/mnt \
            $SCRATCH/random.ext2
```

Figure 10. Command to time and execute test for broadcasting and mounting a software pool.

Secondly, we will measure the time it takes to calculate a hash sum of the test file inside the container after the software pool is copied to local SSDs and mounted using our tools. We only have to run this test on one node since this operation would always happen independently of all the nodes inside a bigger job, as the software is copied to local SSDs on the nodes. The results of this test can then be added to the results of the previous test for any number of nodes. Again, there will be 10 runs, to produce a dependable result. The same test will be conducted, but without accessing the test file through a fuse mount. By comparing these two results, we can isolate the effect that the fuse mount has on our setup's overall performance.

VI. Results

The results of our first benchmark can be seen in Figure 11. In this figure, we compare the performance of the `sw-pool` tool, including the time to calculate a hash sum of the test file over the mount point, to the results from our previous paper. We can see a clear improvement when comparing the new results to the performance of the first implementation of the OSR file system. The software pool tools are almost twice as fast overall: 10.224 seconds against 19.606 seconds with two nodes and 16.159 seconds against 27.866 seconds with

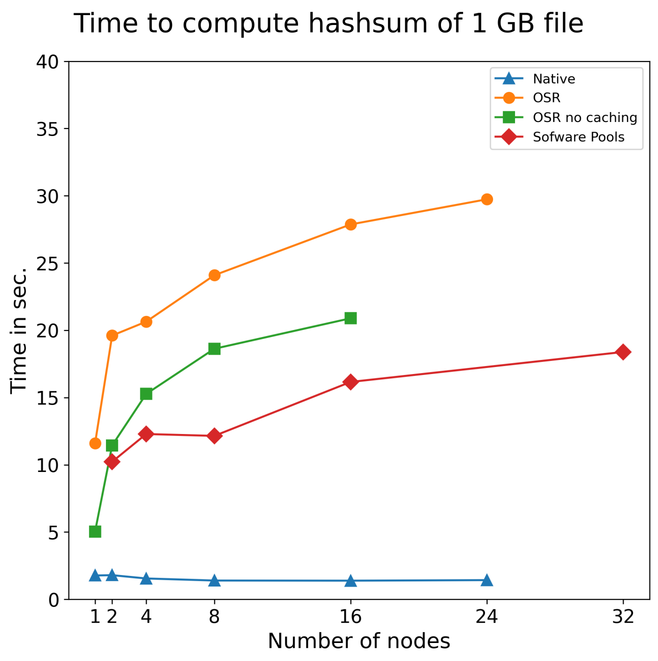

Figure 11. Timings of Software Pool Tool, OSR, OSR without caching and native file system.

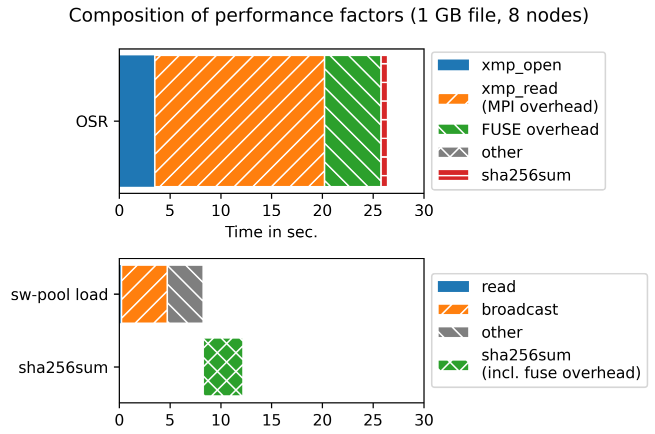

Figure 12. Composition of performance factors for the OSR file system and Software Pools.

16 nodes. With more than 2 nodes, the software pools are almost 25 % faster than the improved OSR implementation (16.159 seconds against 20.889 seconds with 16 nodes). The native scratch file system is still much faster than all the other options.

The results from the performance factor analysis can be seen in Figure 12. Here we compare the results when using software pools against the first implementation of the OSR file system. The biggest difference is clearly the time needed for the communication procedures. We were able to reduce that from 16.711 seconds to 4.512 seconds for handling a 1 GB sized file. The overhead caused by using a fuse also seems slightly improved (3.921 seconds including hash sum calculation against 5.557 seconds excluding hash sum calculation).

VII. Discussion

The software pools and corresponding tools that we presented in this paper showed much-improved performance compared to the OSR file system we presented in our previous paper. This was achieved while still reducing the load put on the storage nodes and network infrastructure to a minimum. The performance increase was achieved by shifting the critical communication procedure to before the user application starts. That allowed us to use MPI's broadcasting method instead of one-sided communication. This change also results in much better latency during runtime, which can lead to an even bigger performance advantage in real-world applications. Especially since with the new approach, existing caching mechanisms in `fuse2fs` and the underlying native file system on the node can have a positive effect, while the OSR file system did not have a working caching mechanism. Additionally, the concept of software pools and the implemented tools are much more mature than the existing implementation of the OSR file system. This was possible since we were able to use existing tools, such as `make2fs` and `fuse2fs` for handling and mounting our software pool image files and `openssl` for signing and verifying our software pools. With our integration into the existing Lmod environment, the software pool tools are almost ready to go into production, while the OSR implementation is missing basic features, such as a working caching mechanism, handling multiple files at once, and multi-threading.

VIII. Conclusion and Future Work

In this paper, we presented a new way to distribute software in HPC environments with our software pools. Software pools offer a simple way to create software environments on HPC systems. By being able to represent whole software stacks in a single file, software pools offer multiple additional benefits. Software pools can be shared easily between users and HPC systems, are reproducible, and reduce the load on stage nodes and network infrastructure. By using a single file, software pools also reduce the usage of Inodes and offer an additional trust factor by being able to sign and verify them with private/public key pairs. The corresponding tools we implemented showed greatly improved performance when compared to the OSR file system from our previous paper. This was achieved by shifting the critical communication procedures to before the user application runs. We were able to seamlessly integrate software pools into the existing module management software Lmod. The tools we presented are already very mature and can go into production without much additional work.

In the future, we want to put software pools into production on the HPC systems hosted by GWDG. To that end, we need to work on documentation, user support, and user training. The implemented tools should also be further improved and expanded. For example, we want to explore other file types, such as squashFS images, that could be used as software pools.

References

[1] J. Dieterle, H. Nolte, and J. Kunkel, "Scalable software distribution for HPC-systems using MPI-based file systems in user space", in *SCALABILITY 2024 : The First International Conference on Systems Scalability and Expandability*, Valencia, Spain, Nov. 2024, pp. 14–20, ISBN: 978-1-68558-216-6.

[2] 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.

[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] 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.

[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] R. Hat, "Gluster", [Online]. Available: https://github.com/gluster/glusterfs (visited on 05/26/2025).

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[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.

[14] R. Pasupuleti, R. Vadapalli, C. Mader, and V. J. Milenkovic, "Apptainer-based containers for legacy and platform dependent machine learning applications on HPC systems", in *2024 IEEE International Conference on Big Data (BigData)*, 2024, pp. 6083–6091. DOI: 10.1109/BigData62323.2024.10825376.

[15] R. McLay, "Lmod: A new environment module system", [Online]. Available: https://lmod.readthedocs.io/en/latest/ (visited on 03/14/2025).

[16] G. für wissenschaftliche Datenverarbeitung mbH Göttingen, "NHR-NORD@Göttingen systeme "Emmy"", [Online]. Available: https://gwdg.de/hpc/systems/emmy/ (visited on 03/14/2025).