Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

A Comparison of Burst Buffer Systems

communicated by Prof. Matthias S. Müller

Bachelorarbeit

Felix Jan Wittlinger
Matrikelnummer: 406692

Aachen, den 19. Oktober 2023

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (’)
Zweitgutachter:	Prof. Dr. Julian Kunkel (*)
Betreuer:	Philipp Martin, M.Sc. (’)

(’) Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Institut für Informatik, Georg-August-Universität Göttingen

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 19. Oktober 2023

Simulations were performed with computing resources granted by RWTH Aachen University under project thes1418.

Kurzfassung

Burst Buffer Systeme bilden eine wichtige Komponente in modern Hochleistungsrechnern, weil die Anzahl wissenschaftlicher Anwendungen und Simulation mit hohen I/O Anforderungen stetig steigt. Burst Buffer schließen die Performancelücke zwischen I/O intensiven Anwendungen und dem parallelen Dateisystem.
Wir haben die zwei Node-lokalen Burst Buffer Dateisysteme BeeOND und GekkoFS getestet. Desweitern haben wir diese qualitativ und quantitativ verglichen und konnten einen durchschnittlichen Vorteil für die I/O Rate bei GekkoFS von 10 % zeigen. Davon hervor stechen die Ergebnisse mit dem NBP BT-IO Benchmark in der `simple-io` Variante, welche kollektives Buffern nicht zulässt. In diesen Tests hatte GekkoFS sogar einen Performancevorteil um den Faktor 10.

Stichwörter: HPC, BeeOND, GekkoFS, Burst Buffer, I/O

Abstract

Burst buffer systems are an important component for modern high-performance computing systems as the count of scientific applications and simulations with high I/O demands grows rapidly. Burst Buffers close the performance gap between I/O intensive computations and the parallel file system.
We tested the two node-local burst buffer file systems BeeOND and GekkoFS. We compared them by qualitative and quantitative aspects, showing an average performance advantage of 10 % in I/O data rates for GekkoFS. Most noticeable are the results for the NPB BT-IO benchmark in the `simple-io` which prevents collective buffering. In these tests GekkoFS outperformed BeeOND up to one order of magnitude.

Keywords: HPC, BeeOND, GekkoFS, Burst Buffer, I/O

Contents

List of Figures

List of Tables

1. Introduction

In high-performance computing (HPC) applications, especially scientific simulations — with high amounts of data read or produced — often suffer from low bandwidths or metadata performance to or from the parallel file system. This issue is mainly caused by the large amount of data processed or produced by these programs. Therefore, there is an increasing need for fast and reliable storage to buffer the data needed by HPC specific applications and simulations. In particular, there is a need for storage systems able to absorb the bursty I/O situations mainly at the beginning or the end of the execution of such applications, while providing high bandwidths and exclusive access. Such that almost now latencies are introduced e.g. by a shared network to the parallel filesystem.
To solve this problem, burst buffers (BB) were invented, the different types of which will be described later (Section 3.3) in detail. Usually, it is distinguished between node-local burst buffer systems — a simplified structure of such is depicted in figure 1.1 — and centralized ones. These systems use the storage provided at the nodes to create mostly short-living, not very large in storage size — compared to a parallel file system in the background—and non-permanent file systems. All data needed for program execution needs to be preloaded from the parallel file system but is then available to the application with higher bandwidth and without interfering with other jobs. On the other hand, produced and further needed data has to be written back to the parallel file system.
A possible workflow could look like the following: First, the burst buffer system is created, and the input data needed is copied/moved to it. Then during the computation which makes use of the data now available from the burst buffer, generated

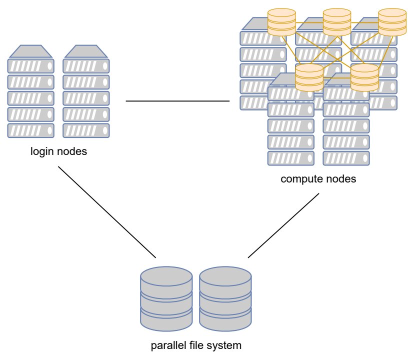

Figure 1.1.: Simplified structure overview of node-local burst buffer. Node-local hard disks / burst buffers are depicted in orange.

data is written to it. After the computation has finished, data that is needed further on needs to be written back to the parallel file system.
Burst buffer systems are often developed and then tested for their presentation paper, in which they are usually compared with similar systems. However, these comparisons often focus on performance only, not necessarily on usability or compatibility with other applications or interfaces such as e.g. POSIX.
Hence, we aim to compare different burst buffer systems and check for their strengths and weaknesses, by evaluating their performance as well as possible use cases. As test candidates, we used BeeOND and GekkoFS which both are node-local BB systems but have substantial internal differences. Through various benchmarks and real-world applications we measured their performance, compared the results and searched for edge cases or challenges influencing their usability.
We found that BeeOND is a good scaling burst buffer filesystem, providing high I/O rates, of up to 3,340 MB/s on 8 Nodes using roughly 80 % of the available accumulated SSD bandwidth. GekkoFS on the other hand scales similarly reaching 3800 MB/s in the same test and therefore performing BeeOND by 12 %.
This thesis is structured as follows. In the next Chapter we present related work, Chapter 3 deals with the background of HPC and burst buffer systems including an explanation of the different burst buffer types. It concludes with a short overview of the tested systems, as well as the benchmark and software used for testing. In Chapter 4 we explain how the tests were executed, followed by presenting their results in Chapter 5. In Chapter 6 we then compare the tested systems. Thus, we discuss the differences seen and evaluate the qualitative differences between the burst buffer systems tested, while also relating this to other — not tested — systems. This is followed by Chapter 7 in which we conclude our findings.

2. Related work

As traditional (parallel) file systems like Lustre [5], or BeeGFS [2] are no longer able to satisfy the high demands of I/O intensive applications and simulations with respect to bandwidth, metadata performance and latencies, burst buffers like presented by Khetawat et al. [12] have become very popular.
Aside from the two presented and tested here — BeeOND [3] and GekkoFS [33], which both implement node-local burst buffer solutions, there are several other approaches. More node-local ones like BurstFS [36] which creates a short-living ephemeral BB filesystem with scalable metadata indexing, or SSDUP [27] having a dynamic approach, writing only parts of data to the burst buffer while the rest is passed to the parallel file system directly. Both of them are presented in detail later in the comparison. Furthermore, there is CHFS [31] using a distributed key-value store.
On the other hand, there are global burst buffers, which mostly use dedicated memory placed between the parallel file system and the compute node, other than the node-local BBs that occupy the memory of the compute note itself.
Examples of these global burst buffers are DataWarp [9] offering to be loaded via a job script, users usually can interact with Data Warp through POSIX APIs, while also a C library provides additional ways of interaction.
Furthermore, there are DDN's IME—tested for example from Schenck al. [26] as well as BurstMem [37] a BB framework on the top of Memcached [20] implementing a log-structured data organization with indexing.

3. Background

In high-performance computing, so-called clusters are used for dealing with large amounts of complex, compute-intensive workloads or (scientific) simulations. These clusters, also called supercomputers, are equipped with multiple nodes, which nowadays consist of multiple central processing units (CPUs) and node-local storage each. The nodes are connected through a network, allowing multiple nodes to communicate and participate in one computation.
Using these resources HPC-systems satisfy the needs for — compared to common PCs — high usage of memory and compute power especially needed in a scientific context (e.g. compute intensive simulations).

3.1. Cluster Operations

To make use of those resources and provide them to the cluster's users, supercomputers work with batch systems. By submitting job scripts, users can interact with the HPC-system and request the desired resources. Depending on the requested resources and compute time, the batch system will schedule the job to the available resources. As the cluster is used by other users as well, there might be a varying delay between submitting a script and the start of its execution. This waiting time is heavily influenced by the current utilization as well as the requested resources and compute time of the job.
One of the biggest advantages of batch systems is that many users can interact simultaneously with the supercomputer, creating and submitting their job scripts. These jobs then run along or after each other, depending on the decisions made by the scheduler, without blocking the whole system, or influencing other user jobs. Furthermore, it is of course possible for one user to submit multiple job scripts at once, to be scheduled and run over time.
By doing so, the user does not need to stay logged in to the cluster or even wait for the execution of jobs to finish, but can instead focus on other tasks and collect the results of the jobs later.

3.2. Cluster and node architecture

In addition to compute nodes, whose purpose is to run the jobs requested by the cluster's users, a cluster usually provides a low number of dedicated login nodes. Using these login nodes, users can submit their scripts without blocking the compute

nodes. The login nodes shall only be used to submit jobs or collect outputs from previous jobs, as well as for building and compiling software for upcoming work. They are not meant to be used for program execution.
Nevertheless, both login and compute notes are equipped similarly — in terms of kinds of the used components, not necessarily in the same amount or computing power.
A typical node consists of CPUs, memory, and an interconnect, additional accelerators are optional. [28]

- **CPUs**: The CPUs — or processors — are responsible for executing the computing process. To satisfy the high demands and no computation power needed by most modern real-world applications, especially in the context of HPC, nodes are equipped with multiple processors (e.g. at the CLAIX 2018 cluster—which was used for this thesis—each node consists of 2 CPUs with a total of 48 cores).

- **Memory**: Without the possibility of storing data, no application can be executed or computation can be done. Hence, nodes are also supplied with local memory. Jobs need to be scheduled on nodes providing at least the required amount of storage.

- **Network**—also called Interconnect: Tasks involving multiple nodes need to be able to communicate with each other and exchange data. For this purpose, low latencies are needed, therefore in this case often HPC specific Networks such as Omnipath [4] or Infiniband [6] are used. Network technologies have a high impact on the performance of a supercomputer.

- **Accelerators commonly graphics processing units (GPUs)**: Accelerators are not a required part for building an HPC node, but most systems provide at least some dedicated compute nodes that are equipped with accelerators. These accelerators can outperform CPUs on special tasks based on their architecture design.

For comparing different HPC systems, different measurements can be used; the most common is the compute power measured in floating point operations per second (FLOPS). Two times a year, a list of the 500 most powerful supercomputers worldwide — the *Top500* — is released [32]. This list is based on the performance archived on the supercomputers using the popular LINPACK benchmark [24]. It measures the compute power by making the system solve a dense system of linear equations. The competitors are allowed to scale the benchmark to their needs and optimize the software used on the clusters. Although not able to show the systems' overall performance (as not one number can), it gives a good estimate for their peak performance [1]. As for now (Top500 list from June 2023), all of these clusters use a Linux distribution as their operating system.
Furthermore, most clusters use a batch scheduler which, as described before, is

needed to execute jobs on clusters. This is done to minimize the idle time of resources as well as ensure that jobs don't access resources that were not allocated to them and may be used by other users.
For long-term storage of the data needed or produced by the jobs, most HPC-systems rely on a parallel file system (PFS) for that purpose, which will be explained in the next section.

As already mentioned, one of the main purposes of a scheduler is to assign resources to the users, and this instance is needed, as all resources of an HPC-system are shared between the active users.
This sharing does not only apply to nodes—respectively compute time but also to the interconnect and the parallel file system for long-term storage.
In doing so, the connection with the PFS has a quite low bandwidth and might be influenced by other jobs. This can slow down the execution of an input and output (I/O) intensive application noticeably as they spend a lot of time doing input and output operations.
To solve this issue, burst buffers were invented. Burst buffers are usually implemented as a smaller file system (FS) close to the nodes which is exclusively available for one program/job during its execution. They mostly are not shared between different programs of the same user or even between users and are mostly placed near the nodes running the respective job. Therefore, BBs provide a much higher bandwidth with lower latencies and protect against disturbances. Usually, the burst buffer file system is only available during the job execution and destroyed afterward, accordingly, all data needed for the program to be executed, or generated during its runtime and needed further on has to be copied from or to the non-volatile PFS respectively [12], [28].

3.3. Kinds of burst buffers

Burst buffers can be categorized into three main groups; Node-local BB, Grouped BB, and Global BB [12], [28]:

- **Node-local Burst Buffer**: The storage used for the file system in node-local BB sits within the nodes. Therefore, through exclusive access to the available bandwidth, the I/O rates scale linearly with the number of nodes. This is only possible because the I/O operations to the BB file system do not conflict with other traffic (e.g. to the PFS).
  Nevertheless, this design creates a single failure domain. Furthermore, shared data between the nodes can be difficult to handle as the nodes need to synchronize or access each other's memory.

- **Grouped Burst Buffer**: Looking at grouped BB, the file systems are placed near groups of nodes, allowing multiple nodes to access. Due to the local network in between the nodes and the storage, there is almost no delay in providing the data. Compared to the node-local BBs, sharing data is simpler to accomplish here. On the other hand, this type of burst buffer can influence the PFS connection, as it at least partly uses the same network for data transfers.

- **Global Burst Buffer**: The third type of burst buffers, Global BBs (also called shared BBs), are deployed on dedicated compute nodes. As a consequence, sharing data between multiple notes within a file system becomes very easy, as the BB system is independent of the parallel file system. Additionally, bursty I/O traffic does not increase the traffic on the general network this way.
  However, the main downside of this solution is that additional hardware is needed, thus creating high costs for servers and their infrastructure. Overall, this kind of burst buffer leads to good resource isolation, as writing to the PFS will not influence computation [35]

3.4. File systems and Benchmarks

In this section, we will take a look at the BB file systems BeeOND and GekkoFS analyzed in this thesis, as well as the benchmarks IOR, mdtest, NPB BT-I/O and m-AIA used for the according tests.

3.4.1. BeeOND

BeeOND[3] was used as it is preinstalled on CLAIX, it is the on-demand variant of BeeGFS [2], the underlying filesystem for the RWTH cluster. BeeOND — like other BB implementations — allows the creation of a short life span burst buffer file system, usually only available for the duration of one job execution. It is transient over multiple nodes and uses the available node-local hard disks, by doing so, it implements a node-local burst buffer FS. As no connection to the parallel file system is established during computation, there is no interference here. All communications made with the PFS are preloading of required data or storing produced data afterward, which needs to be done manually within the job script. Since the script is run exclusively on the respective nodes, BeeOND creates an isolated environment, and can not be disturbed by other users or computaions.
BeeOND creates a new mount point so that data movement with standard operations such as `cp` is possible. The usage of BeeOND is integrable into schedulers such as slurm [29], such that they can find and allocate the needed resources more easily. Because BeeOND is a burst buffer file system, it is not suitable for long-term storage and will be destroyed after job execution.

3.4.2. GekkoFS

The second Burst Buffer file system evaluated in this thesis is GekkoFS [33], which also creates a node-local BB FS. GekkoFS works from the premise that most scientific HPC applications do not need full POSIX compliance [14] and therefore, a relaxed POSIX is sufficient.

GekkoFS uses the node-local storage of the participating nodes. Although not offering full POSIX, GekkoFS ensures the same consistency for operations accessing a specific file, while the consistency of operations on directories is relaxed. It also does not offer a global locking mechanism so applications themselves are responsible for avoiding conflicts (especially such that are caused by overlapping regions).
GekkoFS runs in userspace and relies on Mercury [30] for load balancing, it benefits from a short deployment time as a short-living BB system that is destroyed after the respective job.

3.4.3. Benchmarks

To measure the performance of the file systems presented before, different benchmarks from synthetic ones such as IOR [11] and mdtest [19] over pseudo applications like NPB BT-IO [21] up to the real-world application m-AIA [16] were used.

IOR and mdtest

IOR (Interleaved or Random) is a common benchmark for parallel I/O that is used for testing the performance of PFSs, it offers various interfaces and patterns. It also includes mdtest, which is described later together they are part of the IO-500 benchmark [10], [13] for HPC systems. We are mainly interested in the write bandwidth, meaning the speed of writing data to the BB system, measured here. For synchronization, IOR relies on MPI [11]. It runs on any POSIX compliant platform but requires a fully installed and configured file system implementation. As IOR runs in userspace, it makes a suitable candidate for testing and comparing different parallel file systems as e.g. [17].
Mdtest, on the other hand, specifically evaluates the (peak) metadata performance of parallel file systems and runs as IOR on any POSIX compliant system. Metadata mostly consists of a high number of (in comparison) very small data elements, containing information such as the file name, the last access/change time, ownership, permissions and more. Metadata performance then deals with the question of how a given system handles a high number of such files. We will take the file creation rate, as a measurement in this thesis.
Usually, when using mdtest for measuring a system's performance mdtest is run on several nodes/cores in parallel, which are synchronized over MPI to use the full capacities of the tested FS. It creates a directory tree of configurable depth and offers different kinds of workloads to test with. These workloads also include a file-only test. The measurements achieved by mdtest are comparative as mdtest provides ways to define a standard test (s. I/O-500) [18] and runs in user space.

NPB BT-IO

The NPB (NASA Advanced Supercomputing Division Parallel Benchmark) —specifically the BT-IO benchmark— was used for the experiments conducted in this thesis to get some first real-world-like evaluations based on the pseudo application of a Block-diagonal solver [21].
NPB was first presented in 1993 containing 5 kernels and 3 pseudo applications. For each contained benchmark, NPB defines problem sizes called classes [22]. In the following years, it was extended with more sub-benchmarks and support for larger problem sizes.
In the BT-IO test, each processor is responsible for calculating multiple cartesian subsets. The number of those will increase proportional to the square root of processes participating, causing only square numbers of processes to be allowed in this benchmark. A first approach for this test was described by Fineberg et al. [7], and later formalized in NPB vers 2.4 [38].

m-AIA

M-AIA, formerly known as ZFS, is a collection of several solvers for different physical problems including such for fluid mechanics, tracing the contours of a flame or particle distribution. Here, m-AIA is used as a state of the art, real-world application with high I/O rates. Its strengths are the modularity and direct hybrid coupling of the used solvers. For solving the given problems, m-AIA uses a mesh with adaptive refinement and dynamic load balancing. While details of the grid generation are out of scope for this thesis, it is important to note that grid cells are numbered in order of the respective Hilbert curve (near cells will get near numbers) [34].
For I/O, m-AIA relies on HDF5 [8] and parallel NETCDF [15], the parallelization is done via MPI with asynchronous communication. Some sections are executable on GPUs, but this won't be discussed here, as the focus of this thesis is I/O performance [16].

4. Experimental Setup

This chapter deals with the benchmarks used and especially their configuration including the used parameters.

4.1. System

All benchmarks performed during this thesis were executed on the c18m partition — meaning no accelerators were available or used — of the CLAIX 2018 at RWTH Aachen, running Rocky Linux release 8.8 with kernel version 4.18.0. The nodes in this partition are equipped with 2 Platinum 8160 CPUs running at 2,1 GHz and consist of 48 cores total. There are 192 GB of RAM and 480 GB of SSD storage available at each node. In total, this partition of the cluster provides 1243 nodes, of which a maximum of 8 were used for one benchmark simultaneously. To improve the accuracy of our measurement, nodes were allocated exclusively, even if not all cores on them were used. The maximum 256 cores are sufficient for the tests and comparisons done here. Exceeding this amount would result in more possible combinations of Nodes-Tasks for each benchmark to test, which is outside the scope of this thesis.

4.2. Benchmarks

As NPB only runs on square numbers of (total) processes, we decided to — if applicable — use the same numbers of processes for all benchmarks in this thesis. This should result in a good overview of the behavior of burst buffers in different scenarios. Scalability over multiple nodes and cores is covered as well as possible problems introduced through increased communication. This with a maximum of 8 nodes leads to the following node - total tasks combinations; 1-1, 1-4, 1-16, 2-4, 2-16, 2-64, 4-4, 4-16, 4-64, 8-16, 8-64 and 8-256.

Note: Full sample scripts for each benchmark can be found in the appendix. There is a script for all benchmarks as performed on BeeOND and one (IOR) for GekkoFS, as the changes made from BeeOND to GekkoFS are identical for the other Benchmarks.

4.2.1. IOR and mdtest

To get a first baseline for the performance, we run the IOR and mdtest benchmarks with three different problem sizes each on all node-task combinations. For IOR, we chose total file sizes of 1 GB, 16 GB, and 256 GB. For mdtest due to multiplications with the total numbers of cores and total numbers of tasks, file counts which are powers of two were needed. Because of that, we used file counts of 1024, 131072 and 524288 to again achieve one small, medium and large test, while keeping the benchmarks at a reasonable size.

```
$MPIEXEC <path-to-ior>/ior -t 1m -b 4m -s 4 -i 10 -F -C -e -o $BEEOND/filetest
```

Figure 4.1.: IOR command line for the small dataset with 4 nodes and 16 tasks

IOR

For the IOR benchmark, the executed command is shown in Figure 4.1, while Table 4.1 lists the used parameter variations explained in more detail next. We used a default transfer size of 1 MB and a block size of 4 MB for all benchmarks. The needed adaptations to achieve the desired total amount of data were made with the segment count parameter `-s`.

Table 4.1.: IOR parameters by I/O libary

Parameters	POSIX	MPIIO
number of nodes	1, 2, 4, 8	2
total number of tasks	1, 4, 16, 64, 256	16, 64
transfer size	1 MB	1 MB
blocksize	4 MB	4 MB
overall data amount per test adjusted with segment count	1 GB, 16 GB, 256 GB	16 GB
repetitions	10	10

We applied the parameters `-F`, `-C`, and `-e`, as this turned out to deliver the most reliable results. The `-F` parameter enables a separate file for each process, instead of all processes writing to the same shred file. Even if not looked at in this thesis, we added the `-C` option which results in a much more accurate read performance by reordering the tasks and enforcing MPI processes to read data which was written by neighboring nodes. Additionally, we added the `-e` option to the IOR call, which reduces the effects of caching by enabling the files to be written to the tested file system instead of just being committed to the memory. Therefore, a `fsync()` is executed after all writes have finished, and only after that, the measurement for the write part is stopped [11]. Finally, the `-i 10` parameter was used to execute 10 iterations of the benchmark automatically.
For our tests with GekkoFS we had to remove the `-C` parameter as this caused

problems with the reading phase of IOR. That way now processes are allowed to read data written by themselves. But as that only influences the read phase of the IOR benchmark, it should not be an issue when comparing the respective results for the write performance.

mdtest

We also ran the mdtest benchmarks to see how the file systems handle numerous small files. An exemplary code line is shown in Figure 4.2, Table 4.2 gives a short overview of the used parameters. We used the `-I` parameter to modify the total file count to the desired amount while setting the write and read size for each file to 4096 KB with the `-w` and `-e` options — we chose these file sizes, as that are the ones also used in the Top500 benchmark. Furthermore, we applied the `-z=0` option, which defines the depth of the created directory tree, together with `-L`, with which all files are written at leaf level — (resulting in root level as the depth is set to 0), to ensure that all files are placed directly into the selected filesystem folder and avoid variances due to the usage of directories. Additionally, we used `-R` as a parameter to randomly stat files, instead of zero-files to test for performance on data more than a pure metadata performance test. We ran the mdtest benchmarks with the `-i 10` as well as IOR to also get 10 automated iterations in this case. The -P parameter gives a more detailed output and was therefore used to acquire more information about the measurements.

```
cd $TMP
$MPIEXEC <path-to-mdtest>/mdtest -I=64 -L -z=0 -w=4096 -e=4096 -i 10 -d=$BEEOND/
    filetest -R -P
```

Figure 4.2.: mdtest command line for the small dataset with 4 nodes and 16 tasks

Table 4.2.: mdtest parameters

Parameters	value
I/O API	POSIX
number of nodes	1, 2, 4, 8
total number of tasks	1, 4, 16, 64, 256
filesize (written/read)	4 KB
tree depth	0
file count per test	1024, 132072, 524288
repetitions	10

4.2.2. NPB

NPB offers a series of benchmarks including a block-triangular solver with the possibility to specifically test for I/O (the NPB BT-I/O benchmark). We used this benchmark to get first near-real-world measurements of the I/O performance of the tested burst buffer systems.
We show the lines executed in Figure 4.3, the only parameter needed for here is -n for passing the number of used tasks to mpiexec. NPB itself does not require or offer any parameter for further configuration, because selecting the executable already determines all other configuration parameters (i.e., desired benchmark and class). Because NPB writes to the folder it is called from we created an exclusive directory for each node-task-class combination on the fly in the respective script. We repeated all NPB tests 5 times, since — in terms of the scope of this thesis — this is a reasonable trade-off between execution time and gained information.
We ran the *full-io* as well as the *simple-io* version of the BT-IO benchmark for all Node/Task combinations but decided — due to massive performance differences addressed later — to not perform the Class D Test for `simple-io`. The main difference between these two versions is that full-io allows for collective buffering, whereas `simple-io` does not.

```
cd $BEEOND
mkdir filetest4N16TA
cd filetest4N16TA
$MPIEXEC -n 16 <path-to-npb-executabels>/bt.A.x.mpi_io_full
```

Figure 4.3.: NPB command lines for class A with a total number of 16 tasks

4.2.3. m-AIA

To further test the two file systems considered in this thesis and evaluate their performance with real-world software, we executed m-AIA on them. For our test, we used a benchmark version of m-AIA, which requires 512 tasks. Therefore, 11 nodes were used, as they together provide enough cores. Unfortunately, our tests ran into errors, that could not be resolved so far. We were not able to get any alternative real-world software suitable for benchmarking in the scope of this thesis running until the deadline. Further tests have to be made, and we assume that with enough time and effort, the problems with m-AIA can be solved. Nevertheless, this has to be left for further work.

4.3. Adaptations needed for GekkoFS

To get Gekko running, we needed to add an `-cpus-per-task` option to the job script, this then provides enough resources to execute GekkoFS and the performed benchmarks simultaneously, while GeekoFS does not block the benchmark. The parameter was set in such a way that the `cores-per-task` and `ntasks-per-node` for the benchmark multiplied to 48, which is the number of cores available per node. Due to time limitations and configuration options, we could not repeat the respective tests on BeeOND. Differences shown later might at least partly be caused by this necessary change. Performing these tests and ‘tuning‘ for BeeOND may increase the performance there as well. This needs to be done in further work.

5. Evaluation

In this chapter, we present the results of the measurements taken with the benchmarks presented before. We start with the results archived by BeeOND in the first section and conclude with them from GekkoFS in the second section.

5.1. BeeOND

Taking a look at the overall results of the benchmarks presented in detail later — as one would expect — we see increasing performance with an increasing number of nodes.

5.1.1. IOR

The IOR results as depicted in Figure 5.1 show an approximately linear growth over increasing numbers of cores. With more data to process, it seems as if a utilization especially for the lower node counts is reached. The measured values are approximately within 80 % of the theoretical peak write performance of the SSDs of 510 MB/s [25]. A further (not depicted) test on the 256 GB set with 8 full nodes (8*48 tasks) showed a slight improvement compared to 256 tasks, which is still within the expectations.
In addition to the normal benchmark test, in which we had the `switches` parameter for slurm activated — ensuring getting nodes closer together — we verified this here for IOR on BeeOND by performing that same test without the `switches` parameter. Without this parameter and therefore, not enforcing nearby nodes we saw similar, but less stable measurements, with higher variations. Hence, we decided to perform all further tasks only with the `switches` parameter activated.

IOR with MPI-IO

To not exclusively run all tests on POSIX-IO and find possible weaknesses, we repeated the IOR measurements for 2 nodes and 16 tasks as well as for 2 nodes and 64 tasks with MPI-IO. A comparison with the respective values out of the run with POSIX is shown in Figure 5.2. As depicted there, for BeeOND these two deliver about the same results in respect to write performance.

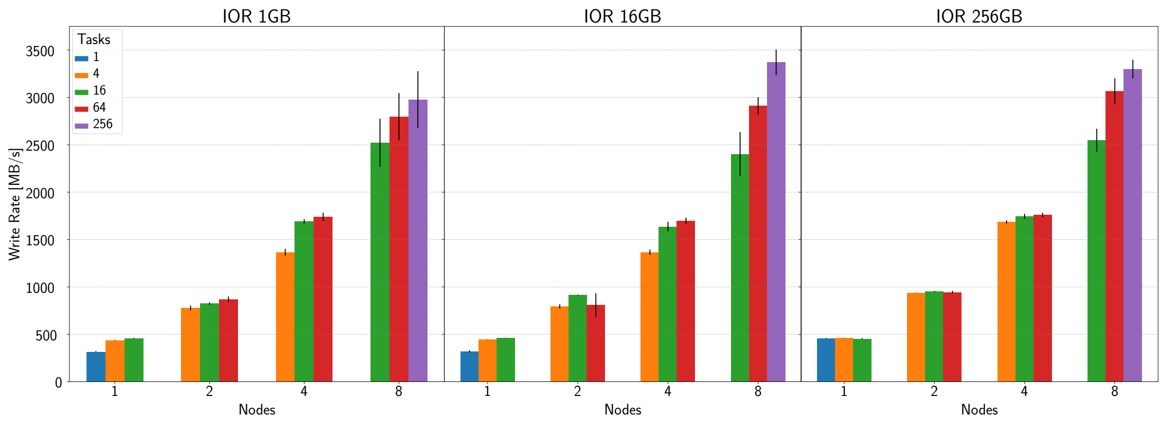

Figure 5.1.: **IOR results on BeeOND** On the y-axis, write rate in Megabytes per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviation calculated over the 10 iterations.

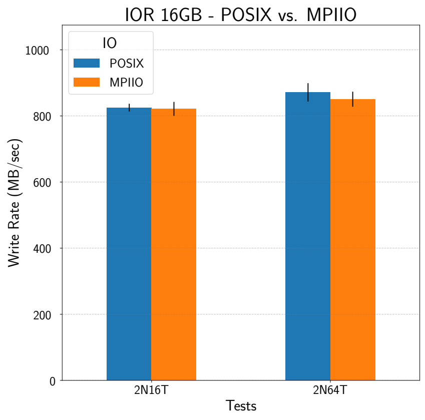

Figure 5.2.: **IOR MPI-IO vs. POSIX-IO on BeeOND** On the y-axis, write rate in Megabytes per second is depicted, the x-axis shows the tested node(N) and task(T) combinations. The bar colors correspond to the use IO scheme. Error bars show the standard deviation calculated over the 10 iterations, for each test.

5.1.2. mdtest

Mdtest shows similar results over all data sets, reaching an upper bound at a file creation rate of approximately 8200. Although there are (a) some drops in the 1024 file test at 8 nodes as well as (b) 4 nodes with 4 total tasks for all test sizes, these can be explained with either (a) too few files per process or (b) too wide of a spread of files with an in comparison low computation power. Especially for the large test it seems that with a file creation rate of about 8200, a saturation is reached. With one node the measured performance is notably lower, which we assume to be caused by only being able to access one SSD.

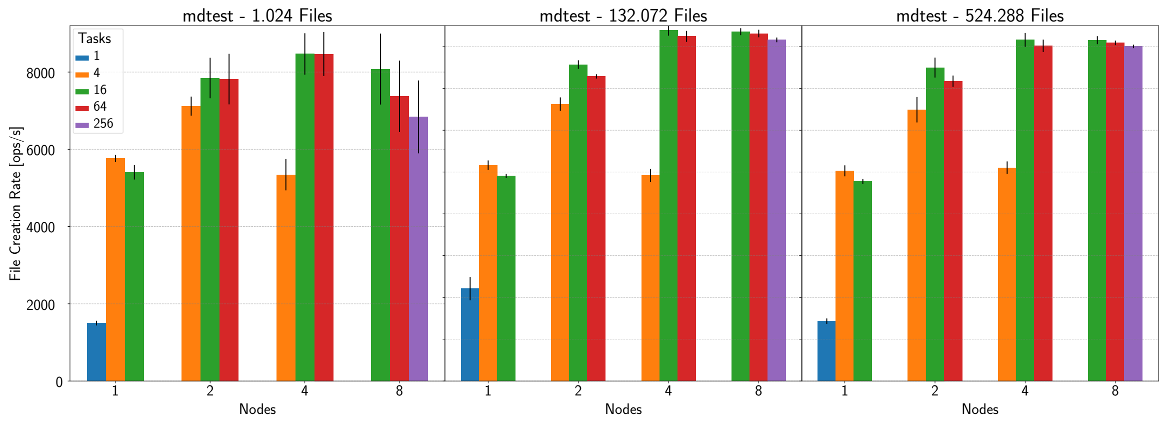

Figure 5.3.: **mdtest results on BeeOND** On the y-axis, write rate in file creation rate per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviation calculated over the 10 iterations.

5.1.3. NPB

In the following, we present the results of the NPB BT-IO benchmark executed on BeeOND. While on `full-io` we overall notice increasing performance for higher node counts, in the `simple-io` benchmark the performance suffers a lot.

full-io

The measurements taken with the NPB benchmark in the `full-io` version, performed as a first near-real-world test, show similar results as the IOR benchmark. The performance is mostly increasing with the task count — best shown in Class D. We notice, that for all dataset sizes the test with 256 tasks gives significantly lower I/O rates and assume that is based on the increasing need for communication between the used cores. Also noticeable are decreased measurements for 64 tasks compared with 16 tasks at 2 and 4 nodes on the small — and medium-sized tests. The Test in class D, with 1 Task, is not depicted, as it ran into an error showing `Error writing to file` shortly after starting in all our executions. The reason for that issue remains unclear and might be a possible aspect of further work.

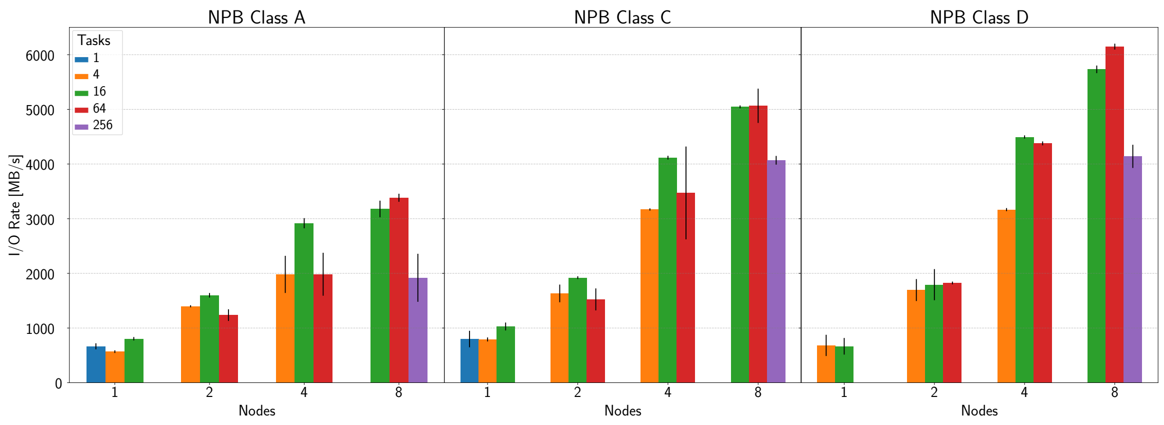

Figure 5.4.: **NPB BT-IO results on BeeOND (full-io):** On the y-axis, I/O rate in Megabytes per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviations calculated over the 5 iterations.

simple-io

Now looking at the measurements taken with the `simple-io` version of NPB BT-IO, shown in Figure 5.5, we see a decline in the size of two orders of magnitude. This difference caused by not allowing the operating processes to use collctive buffering, is enormous. Therefore, the influence of this on BeeOND can not be neglected and must be kept in mind by the user. As before in the `full-io` test more cores lead to more communication needs, resulting in further declines for the with 4 and 8 nodes.

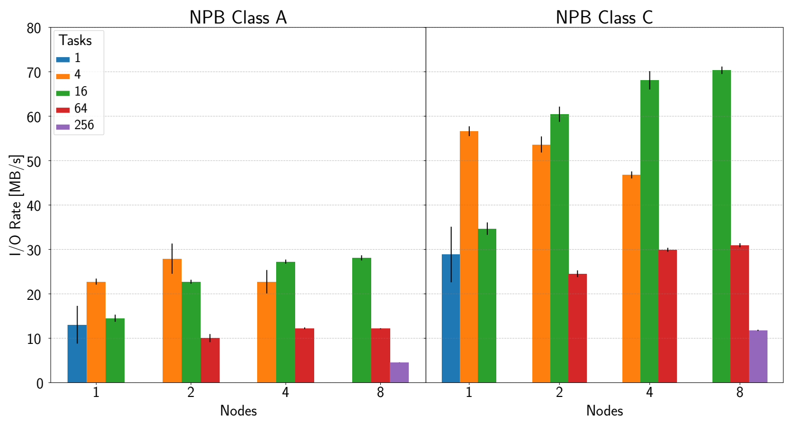

Figure 5.5.: **NPB BT-IO results on BeeOND (simple-io):** On the y-axis, I/O rate in Megabytes per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviations calculated over the 5 iterations.

5.2. GekkoFS

Similar to BeeOND the benchmark results for GekkoFS presented in the following, mainly show a performance increase with increasing numbers of nodes.

5.2.1. IOR

We present the IOR results for GekkoFS in Figure 5.6, these show a linear growth over increasing node counts reaching 80 % - 90 % of the theoretical SSD performance. Over the changing test sizes, especially the smaller node counts reach a saturation. The measurements for the 8 node test, nevertheless show some differences for commuting the benchmark.

IOR with MPI-IO

Executing IOR with MPI-IO in the reduced data set, as previously described for BeeOND, on GekkoFS results in an error as shown in Figure 5.7. Based on this error we conclude, that GekkoFS at least in the version tested here (v0.91) is not capable of working with the MPI-IO interface, which limits its overall use cases.

5.2.2. mdtest

While running the mdtest benchmarks on GekkoFS we ran into issues, namely non-deterministic behavior during the start-up. Running the same script multiple times, either mdtest was not started at all, or it started but did not provide any results in a reasonable time. Nonetheless, the few results archived, before stopping our test and

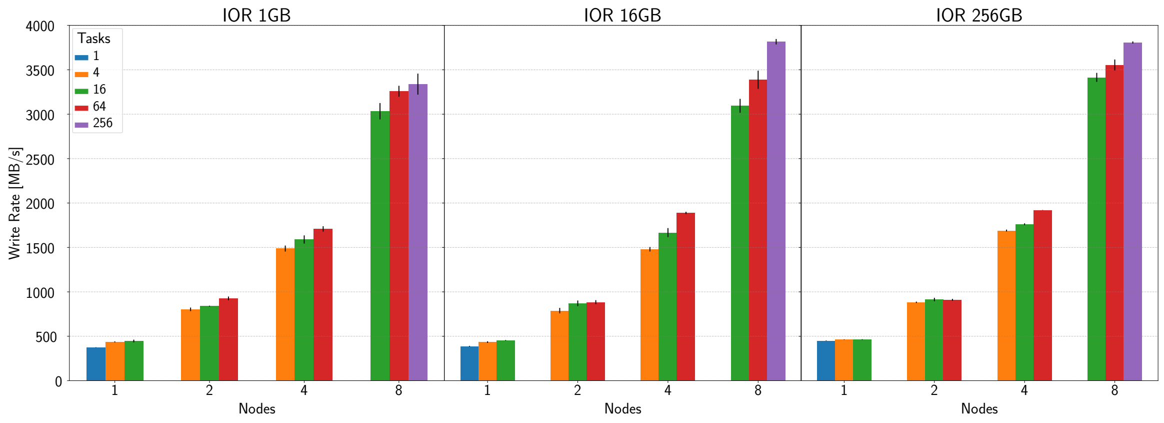

Figure 5.6.: **IOR results on GekkoFS** On the y-axis, write rate in Megabytes per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviation calculated over the 10 iterations.

```
  This requires fcntl(2) to be implemented. As of 8/25/2011 it is not. Generic
      MPICH Message: File locking failed in ADIOI_GEN_SetLock(fd 2710,cmd
      F_SETLKW64/7,type F_WRLCK/1,whence 0) with return value FFFFFFFF and errno
      5F.
- If the file system is NFS, you need to use NFS version 3, ensure that the lockd
     daemon is running on all the machines, and mount the directory with the '
    noac' option (no attribute caching).
- If the file system is LUSTRE, ensure that the directory is mounted with the '
    flock' option.
ADIOI_GEN_SetLock:: Operation not supported
ADIOI_GEN_SetLock:offset 0, length 1048576
Abort(1) on node 5 (rank 5 in comm 0): application called MPI_Abort(
    MPI_COMM_WORLD, 1) - process 5
```

Figure 5.7.: **Error message** produced bei IOR when run with MPIIO on GekkoFS.

efforts to end this nondeterministic behavior, show high variaons, which may also be caused by the problems described before. Based on these issues, we unfortunately can not present any benchmark results here. Investigating these matters in more detail and archiving reliable results need to be investigated further on.

5.2.3. NPB

In the following, we present the results of the NPB BT-IO benchmarks performed on GekkoFS. Unfortunately based on the added `cpus-per-task` parameter, we were not able to perform the NPB Benchmark with 8 nodes, because of resource limitations. These tests, therefore, are left for further work. We would expect to see a similar behavior compared to the test with a smaller number of nodes.

full-io

Our measurements on GekkoFS taken with NPB BT-I/O with the `full-io` version are depicted in Figure 5.8. We, despite having high variations, see an approximately linear growth with increasing node counts. Taking the shown standard deviations into account the measured performances on the same node with different task counts are within 10 %. Some of these variations may be caused by the changing count of available cores per task. Produced by the need introduction of the `cpus-per-task` parameter. Only the test on 1 Task strikes out here.
The test with 1 Task for class D again stops, early in the execution with the same error message as on BeeOND and therefore no values are shown here.

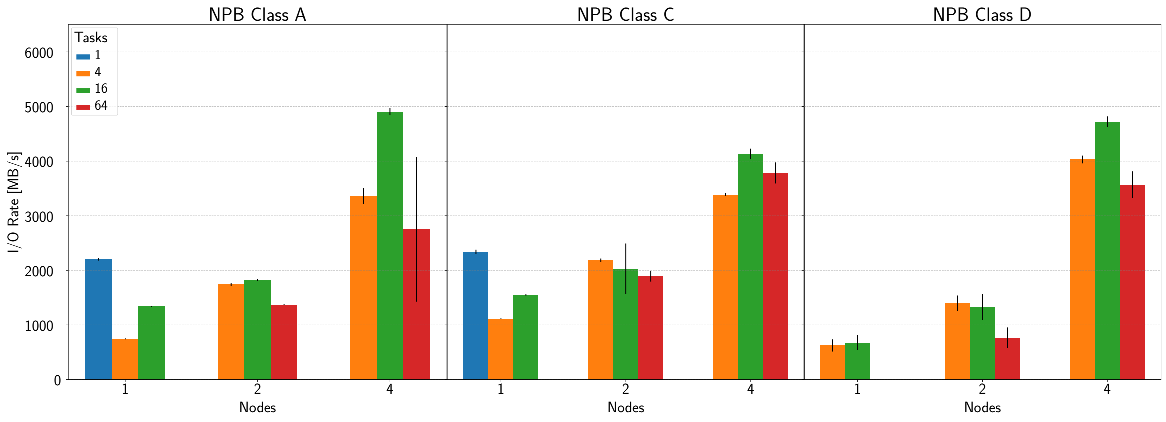

Figure 5.8.: **NPB BT-IO results on GekkoFS (full-io—with collective buffering):** On the y-axis, I/O rate in Megabytes per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviations calculated over the 5 iterations.

simple-io

The measurements with the `simple-io` version — shown in Figure 5.9 — reveal a similar behavior as previously presented for the `full-io` version. The performance grows with an increasing number of nodes almost linearly, while having high variaons.
For almost all tests on the same node count the performance decreases with increasing tasks. Also in comparison to the `full-io` test, the I/O rates archived here are significantly lower. These two phenomenons appear to be caused by the `simple-io` test not allowing collective buffering.
Similar to before with the `full-io` test, the variances and deviation depicted, as well as the performance drops with more tasks per node, could be caused by introducing the `cpus-per-task` parameter.

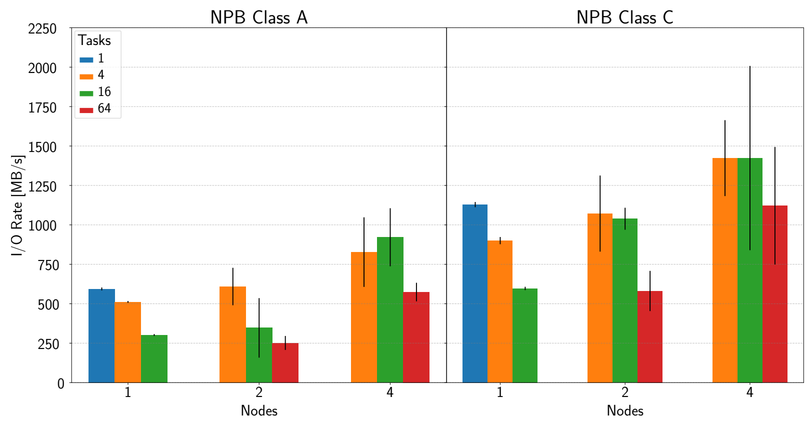

Figure 5.9.: **NPB BT-IO results on GekkoFS (simple-io—without collective buffering):** On the y-axis, I/O rate in Megabytes per second is depicted, the x-axis shows the number of nodes used. The bar colors correspond to the number of tasks. Error bars show the standard deviations calculated over the 5 iterations.

6. Comparison

Taking a look at the two systems GekkoFS and BeeOND tested here, we find that in the end, both systems share some similarities in reaching the aim of creating a burst buffer file system. Gekko and BeeOND both create a node-local, short-living — mostly for the duration of a job — file system. These created BB systems provide exclusive access to the data, and therefore, prevent interferences caused by other jobs or users. Furthermore, through being independent of the parallel file system the BB systems offer high bandwidths for providing the preloaded data or storing data produced by a job.
Considering how that aim described above is achieved, GekkoFS and BeeOND show a very different approach to this.

6.1. Comparing Burst Buffer Systems

6.1.1. GekkoFS

GekkoFS is a standalone file system, that needs to be installed manually by its users, but this is no bigger issue, as it runs in user space.
GekkoFS then needs to be started during the job script if needed, these deploying — according to the developers of GeekoFS on 512 nodes — needs about 20 seconds [33]; while not specifically testing for it in our tests with up to 256 cores the deployment time was well below 30 seconds.
Furthermore, GekkoFS only uses a relaxed POSIX compliance, as it relies on the premise that a full POSIX compliance is not needed for most HPC applications [14]. Users need to keep these restrictions in mind when executing software with a GeekoFS burst buffer file system. Nevertheless, it is as consistent on operations accessing a specific file as a full POSIX compliant system, while operations on directories aren't covered by the relaxed POSIX compliance. As Gekko does not offer a global locking mechanism applications and users are responsible for ensuring not to cause conflicts because of accessing overlapping regions.
Move or rename operations during the execution of a parallel job are not supported too, as they also fall in the category of rarely used functions in HPC programs. To reduce the complexity of the FS in Gekko all file system operations are executed synchronously, without any kind of caching.

6.1.2. BeeOND

BeeOND on the other hand is the on-demand version of BeeGFS and comes with such an installation. Therefore, it can not be installed by the user onto an existing HPC system, but can be included in schedulers as slurm.
BeeGFS as well as GekkoFS creates a node-local burst buffer file system, but in contrast, guarantees full POSIX compliance, so that users do not need to worry about most of the challenges described above.
Additionally, there are numerous other burst buffer systems, all differentiating in lager or minor details on how the BBs are implemented, what premises are made (e.g. regarding POSIX with Geeko), and where the BB FSs are placed within the system.
We will now take a short look at some more node-local ones as well as such not using the node-local storage.

6.1.3. Other node-local Burst Buffer systems

One other node-local burst buffer system is BurstFS [36]. It also creates a short-living (duration of a job) BB FS, that offers scalable metadata indexing and linear scalability of aggregated I/O. Similarly to Gekko, it is constructed on the fly using the node's local SSDs. All allocated and used resources and nodes are cleaned up after execution to avoid post-mortem interferences. The BB system created by BurstFS is exclusive, and no outside job can access it, nevertheless, parallel programs within the same job allocation can share data through it.

Furthermore, there is SSDUP [27] a 'traffic aware burst buffer' system, which is based on OrangeFS [23]. It also used the node's local memory to create a burst buffer solution, but in contrast to the systems presented before not all data written during the communication is stored on the BB itself. SSDUP uses a traffic detector to differentiate between random and other writes, the latter of which are deemed sequentially to the PFS directly, while the random ones are buffered to the BB FS. The random writes to the burst buffer are later converted to sequential ones using the additionally created log structure and then written to the PFS. To avoid being unable to receive data from random writes, the buffer is divided into halves, one half can continue receiving random writes, while the other half flushes the previously received data to the PFS.

6.1.4. Non node-local Burst Buffer system

DataWrap [9] is one system that in comparison to BeeOND, GeekoFS, BurstFS, and SSDUP uses dedicated burst buffer nodes placed between the compute nodes and the parallel file system.
As with BeeOND, to use DataWrap this needs to be requested with according directives in the job script. Users can make use of DataWrap through POSIX APIs,

but additionally, a C library provides a way for other programs to interact and use DataWarp. It provides the possibility for a compute note swap, meaning that different parts of the application (e.g. pre-/postprocessing or computation) can be done by different nodes can be used without moving the corresponding data. This is an option that on node-local burst buffer systems would at least be hard if not impossible implement.
Furthermore, DataWrap offers two types of instances:

- The first type, called *job instance*, works similar to a BeeOND or GeekoFS instance. It offers exclusive access to the BB FS ensuring, that no unallowed access from outside the participating nodes is made and is destroyed after the job has finished

- Secondly there are so-called *persistent instances*, these are not bound to the lifetime of a job. Any job can request access to these by providing the name of the instance. File access nevertheless, will be authorized using POSIX file permissions.
  These instances are usually explicitly created outside a job.

Aftertermination of an instance DataWarp ensures, that further needed data will be flushed to the PFS.

6.2. Comparison of the archived performances

In this section, we will compare the measured performances archived by BeeOND and GekkoFS, during our test.

We observe that for the IOR benchmark BeeOND and GekkoFS show a similar behavior, of linear growth over increasing node counts. Both reach about 80 % of the discs' specification in the small test climbing to 90 % with increasing amounts of data. This seems to be a saturation for our tests. Especially for the high node counts on the 256 GB test, GekkoFS outperformed BeeOND by 12%. On the lower node counts the differences might be outside the calculated standard deviation with a small advantage for GekkoFS, but still can be considered irrelevant. The overall performance gains, especially noticeable in the tests with 8 Nodes, are a benefit of GekkoFS and probably result in the relaxed POSIX constraints.

As explained before we were unable to archive reliable results for mdtest on GeekoFS, we can not draw a comparison to the performances measured on BeeOND here. These measurements and comparisons are left for further work. In general, we assume based on what we have seen during our test, despite having high variations in that tests, that GekkoFS at least can provide a similar performance to that measured for BeeOND.
As depicted above, the measurements on BeOND and GekkoFS taken with NPB in

the `full-io` version show a similar pattern, with few exceptions. GekkoFS hereby mostly has the advantage over BeeOND, while on the other side, it shows higher variations.
Nevertheless, GekkoFS handles the `simple-io` benchmark in our tests a lot better than BeeOND. BeeOND for these measurements suffered a performance loss of two orders of magnitude, while Gekko only had a decline of factor 5. Additionally, GekkoFS showed an increase in performance when using multiple notes, while in the same test BeeOND no such increase is noticeable.

6.3. Findings

As we have shown before, GekkoFS can outperform BeeOND in the tested configuration in I/O performance, averaging by a performance overhead of 10-15 %. For special use cases such as I/O with non-collective buffering, this advantage even reaches one order of magnitude. Nevertheless, these performance gains could at least partly be archived by the introduced `cpus-per-task` parameter. Further investigations and efforts have to show if BeeOND can achieve similar performance.
We consider GeekoFS — based on our previously described experiences and the results archived in the IOR and NPB benchmarks — to be a reasonable alternative BB file system. Nevertheless, to be fully usable the problems described have to be solved. We assume, that the issues mentioned before for mdtest are not caused by GekkoFS itself and therefore are a configuration problem of the various parameters. In our tests, GekkoFS was not able to execute IOR with the MPI-IO interface, which limits the usability at least slightly as some HPC applications offer MPI-IO exclusively. Regardless of this issue, during our tests, we were not able to show any specific problems when using GekkoFS, that could be related to the use of relaxed POSIX compliance.

7. Conclusion

We have tested the two burst buffer systems BeeOND, which is the on-demand variant of BeeGFS, and GekkoFS a standalone BB system running in user space. While both create a node-local burst buffer file system, one main difference is that GekkoFS relies on a relaxed POSIX compliance, while BeeOND offers full POSIX compliance.
In our test, GekkoFS archived a performance overhead compared to BeeGFS of 8-10 %. Very promising is the performance gain seen in the NPB BT-IO test with `simple-io`, archiving results about one order of magnitude greater than the tests on BeeOND. Nevertheless, it can not be neglected that we were only able to perform any tests on GekkoFS after introducing the `cpus-per-task` parameter. It remains to be investigated if adding this parameter had an influence on the performance of GekkoFS and if through appropriate tuning BeeOND can reach similar results.
Although GeekoFS had problems with the MPI-IO interface in our test which could at least slightly limit the usability, we had—without specifically testing for it— no issues with the related POSIX constraints.
When the mentioned variances are minimized and the explained problems are solved we expect GekkoFS to be a considerable alternative burst buffer system. Especially, if the described performance overheads survive, these give an advantage for POSIX-IO-driven HPC applications.

A. Apendix

A.1. BeeOND Scripts

```
#!/usr/local_rwth/bin/zsh

#SBATCH --beeond
#SBATCH --exclusive
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=16
#SBATCH --job-name=beeond_ior_1G_4N_64T
#SBATCH --output=<path-to-outputfile>/beeond_ior_1G_4N_64T_%J.txt
#SBATCH --switches=1

#SBATCH --time 00:15:00
#SBATCH --account=thes1418

cd $TMP
$MPIEXEC <path-to-ior>/ior -t 1m -b 4m -s 4 -i 10 -F -C -e -o $BEEOND/filetest
```

Figure A.1.: IOR sample Code BeeOND

```
#!/usr/local_rwth/bin/zsh

#SBATCH --beeond
#SBATCH --exclusive
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=16
#SBATCH --job-name=beeond_md_1K_4N_16T
#SBATCH --output=<path-to-outputfile>/beeond_md_1K_4N_16T_%J.txt
#SBATCH --switches=1

#SBATCH --time 00:15:00
#SBATCH --account=thes1418

cd $TMP
$MPIEXEC /<path-to-mdtest>/mdtest -I=64 -L -z=0 -w=4096 -e=4096 -i 10 -d=$BEEOND/
    filetest -R -P
```

Figure A.2.: mdtest sample Code BeeOND

```
#!/usr/local_rwth/bin/zsh

#SBATCH --beeond
#SBATCH --exclusive
#SBATCH --nodes=4
#SBATCH --ntasks-per-node=16
#SBATCH --job-name=beeond_NPB_A
#SBATCH --output=<path-to-outputfile>/beeond_NPB_A_4N_16T_%J.txt
#SBATCH --switches=1

#SBATCH --time 00:20:00
#SBATCH --account=thes1418

cd $BEEOND
mkdir filetest4N16TA
cd filetest4N16TA
$MPIEXEC -n 16 <path-to-npb-execuatabels>/bt.A.x.mpi_io_full
```

Figure A.3.: NPB sample Code BeeOND

A.2. Gekko Script

```
#!/usr/local_rwth/bin/zsh

#SBATCH --exclusive
#SBATCH --output=<path-to-outputfile>/gekko_iorS_4N16T--%J.txt
#SBATCH --account=thes1418
#SBATCH --cpus-per-task=12
#SBATCH --nodes=4
#SBATCH --time 00:05:00
#SBATCH --switches=1

#Setup environment
source <path-to-file>/variables.source

export LIBGKFS_HOSTS_FILE=<path-to-hostfile>/Hostfile_4N16T.TMP
export GEKKO_LIB=<path-to-gekko-install-dir>/install/lib64/libgkfs_intercept.so
export GEKKO_DAEMON=<path-to-gekko-install-dir>/install/bin/gkfs_daemon
export GEKKOFS=$TMP/gekko_mount

#Start Gekko Daemons
$MPIEXEC $FLAGS_MPI_BATCH --overlap --ntasks-per-node 1 $GEKKO_DAEMON -r $TMP -
    m $GEKKOFS -H $LIBGKFS_HOSTS_FILE -l ib0 -c --auto-sm &

#busy waiting until all daemons are registered
until [[ "$(wc␣-l␣${LIBGKFS_HOSTS_FILE}␣2>␣/dev/null␣|␣awk␣'{print␣$1}')" != ${
    SLURM_JOB_NUM_NODES} ]]; do
        sleep 1
done

#give the hostfile enough time to be synchronized (without this, sometimes the
    file is not found by ior)
sleep 3

#Run ior test
$MPIEXEC --export LD_PRELOAD=${GEKKO_LIB} --export LIBGKFS_HOSTS_FILE
    $FLAGS_MPI_BATCH --ntasks-per-node 4 --oversubscribe <path-to-ior>/ior -t 1
    m -b 4m -s 16 -i 10 -F -e -o $GEKKOFS
```

Figure A.4.: IOR sample Code GekkoFS

Bibliography

[1] [Online]. Available: `https://top500.org/project/linpack/` (visited on 10/05/2023).

[2] *BeeGFS Documentation 7.4.1.* [Online]. Available: `https://doc.beegfs.io/latest/index.html` (visited on 10/08/2023).

[3] *BeeOND: BeeGFS On Demand - Documentation.* [Online]. Available: `https://doc.beegfs.io/latest/advanced_topics/beeond.html` (visited on 07/15/2023).

[4] M. S. Birrittella *et al.*, "Intel® omni-path architecture: Enabling scalable, high performance fabrics", in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 1–9. DOI: `10.1109/HOTI.2015.22`.

[5] P. J. Braam, "Lustre: A Scalable, High-Performance File System", [Online]. Available: `https://cse.buffalo.edu/faculty/tkosar/cse710/papers/lustre-whitepaper.pdf` (visited on 10/08/2023).

[6] R. Buyya, T. Cortes, and H. Jin, "An introduction to the infiniband architecture", in *High Performance Mass Storage and Parallel I/O: Technologies and Applications*. IEEE, 2002, pp. 616–632. DOI: `10.1109/9780470544839.ch42`.

[7] S. A. Fineberg *et al.*, "Pmpio - a portable implementation of mpi-io", in *Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation*, ser. FRONTIERS '96, USA: IEEE Computer Society, 1996, p. 188, ISBN: 0818675519.

[8] M. Folk and E. Pourmal, "Balancing performance and preservation lessons learned with HDF5", in *Proceedings of the 2010 Roadmap for Digital Preservation Interoperability Framework Workshop*, Gaithersburg, Maryland, USA: Association for Computing Machinery, 2010, ISBN: 9781450301091. DOI: `10.1145/2039274.2039285`.

[9] D. Henseler *et al.*, "Architecture and Design of Cray DataWarp",

[10] V. for I/O, *IO-500 - web.* [Online]. Available: `https://www.vi4io.org/std/io500/start` (visited on 07/28/2023).

[11] *IOR documentation - readthedocs*, en. [Online]. Available: `https://buildmedia.readthedocs.org/media/pdf/ior/latest/ior.pdf` (visited on 07/28/2023).

[12] H. Khetawat *et al.*, "Evaluating burst buffer placement in hpc systems", en, in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, Albuquerque, NM, USA: IEEE, 2019, pp. 1–11, ISBN: 978-1-72814-734-5. DOI: `10.1109/CLUSTER.2019.8891051`.

[13] J. M. Kunkel *et al.*, "Establishing the IO-500 benchmark", *White Paper*, 2016.

[14] P. H. Lensing *et al.*, "File system scalability with highly decentralized metadata on independent storage devices", in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, ser. CCGRID '16, Cartagena, Columbia: IEEE Press, 2016, pp. 366–375, ISBN: 978-1-5090-2452-0. DOI: `10.1109/CCGrid.2016.28`.

[15] J. Li *et al.*, "Parallel NetCDF: A high-performance scientific I/O Interface", in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03, Phoenix, AZ, USA: Association for Computing Machinery, 2003, p. 39, ISBN: 1581136951. DOI: `10.1145/1048935.1050189`.

[16] A. Lintermann, M. Meinke, and W. Schröder, "Zonal Flow Solver (ZFS): A highly efficient multi-physics simulation framework", *International Journal of Computational Fluid Dynamics*, vol. 34, no. 7-8, pp. 458–485, 2020, ISSN: 1061-8562. DOI: `10.1080/10618562.2020.1742328`.

[17] *Lustre wiki - ior.* [Online]. Available: `https://wiki.lustre.org/IOR` (visited on 07/28/2023).

[18] *Lustre wiki - mdtest.* [Online]. Available: `https://wiki.lustre.org/MDTest` (visited on 07/28/2023).

[19] *Mdtest*, 2023. [Online]. Available: `https://github.com/LLNL/mdtest` (visited on 07/28/2023).

[20] *Memcached - a distributed memory object caching system.* [Online]. Available: `https://www.memcached.org/` (visited on 10/08/2023).

[21] *NAS Parallel Benchmarks.* [Online]. Available: `https://www.nas.nasa.gov/software/npb.html` (visited on 07/28/2023).

[22] NASA, *Problem sizes and parameters in nas parallel benchmarks.* [Online]. Available: `https://www.nas.nasa.gov/software/npb_problem_sizes.html` (visited on 07/28/2023).

[23] *OrangeFS.* [Online]. Available: `https://www.orangefs.org/` (visited on 10/08/2023).

[24] A. Petitet *et al.* [Online]. Available: `https://www.netlib.org/benchmark/hpl/` (visited on 10/05/2023).

[25] *Product Brief SSDs.* [Online]. Available: `https://www.mouser.de/pdfdocs/Intel_SSD_D3-S4510_D3-S4610_PB_337756-001-1385547.pdf` (visited on 10/04/2023).

[26] W. Schenck *et al.*, "Early Evaluation of the "Infinite Memory Engine" Burst Buffer Solution", in *High Performance Computing*, M. Taufer, B. Mohr, and J. M. Kunkel, Eds., vol. 9945, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2016, pp. 604–615, ISBN: 978-3-319-46078-9 978-3-319-46079-6. DOI: `10.1007/978-3-319-46079-6_41`.

[27] X. Shi *et al.*, "SSDUP: A traffic-aware ssd burst buffer for HPC systems", in *Proceedings of the International Conference on Supercomputing*, ser. ICS '17, New York, NY, USA: Association for Computing Machinery, 2017, pp. 1–10, ISBN: 978-1-4503-5020-4. DOI: `10.1145/3079079.3079087`.

[28] R. Simons, *Parameterevaluation von dateisystemen in hochleistungsrechnen*, de, Masterthesis -RWTH, Aug. 2022.

[29] *Slurm Workload Manager - Documentation.* [Online]. Available: `https://slurm.schedmd.com/` (visited on 10/08/2023).

[30] J. Soumagne *et al.*, "Mercury: Enabling remote procedure call for high- performance computing", in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, ISSN: 2168-9253, 2013, pp. 1–8. DOI: `10.1109/CLUSTER.2013.6702617`.

[31] O. Tatebe *et al.*, "CHFS: Parallel Consistent Hashing File System for Node-local Persistent Memory", in *International Conference on High Performance Computing in Asia-Pacific Region*, ser. HPCAsia2022, New York, NY, USA: Association for Computing Machinery, 2022, pp. 115–124, ISBN: 978-1-4503-8498-8. DOI: `10.1145/3492805.3492807`.

[32] *Top500.* [Online]. Available: `https://www.top500.org` (visited on 06/27/2023).

[33] M.-A. Vef *et al.*, "Gekkofs - a temporary distributed file system for hpc applications", in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2018, pp. 319–324, ISBN: 978-1-5386-8319-4. DOI: `10.1109/CLUSTER.2018.00049`.

[34] M Waldmann *et al.*, *Implementation of a lattice Boltzmann method on GPU based HPC Systems*, m-aia presentation slides. [Online]. Available: `https://indico3-jsc.fz-juelich.de/event/35/contributions/36/attachments/22/29/Waldmann_2022_slides_ImplementationOfALatticeBoltzmannMethodOnGpuBasedHpcSystems.pdf` (visited on 08/08/2023).

[35] T. Wang, *Exploring novel burst buffer management on extreme-scale hpc systems*, 2017. [Online]. Available: `https://diginole.lib.fsu.edu/islandora/object/fsu3A507737/`.

[36] T. Wang *et al.*, "An ephemeral burst-buffer file system for scientific applications", in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE Press, 2016, pp. 807–818. DOI: `10.1109/SC.2016.68`.

[37] T. Wang *et al.*, "BurstMem: A high-performance burst buffer system for scientific applications", en, in *2014 IEEE International Conference on Big Data (Big Data)*, Washington, DC, USA: IEEE, 2014, pp. 71–79, ISBN: 978-1-4799-5666-1. DOI: `10.1109/BigData.2014.7004215`.

[38] P. Wong and R. F. V. der Wijngaart, *NAS Parallel Benchmarks I/O Version 2.4*. [Online]. Available: `https://www.nas.nasa.gov/assets/nas/pdf/techreports/2003/nas-03-002.pdf`.