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

Parameterevaluation von Dateisystemen in Hochleistungsrechnen

HPC File System Parameter Evaluation

Masterarbeit

Ruben Simons
Matrikelnummer: 409646

Aachen, den 25. Oktober 2022

Communicated by Prof. Matthias S. Müller

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

(’) Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Institute of Computer Science, Georg-August-Universität Göttingen

Simulations were performed with computing resources granted by RWTH Aachen University under project thes1146.

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 25. Oktober 2022

Kurzfassung

Um optimale Performance für Anwendungen im Bereich High Performance Computing sicherzustellen, ist ein schnelles, paralleles Dateisystem von Bedeutung. Deshalb wurden im Rahmen dieser Arbeit die Auswirkungen von Dateisystemparametern auf die Performance von Anwendungen untersucht. Hauptsächlich wurden dabei die Parameter des BeeGFS Dateisystems, welches jeweils für die Dauer der Ausführung einer Anwendung erstellt wurde, variiert. Ebenfalls wurden Tests auf einem zentralen LUSTRE Dateisystem durchgeführt. Anhand von fünf Benchmarks wurden Tests mit bestimmten Parameterkombinationen durchgeführt, um die Performance des Dateisystems für unterschiedliche Anwendungsfälle zu testen.

Betrachtet wurden hierbei für BeeGFS Clientparameter, Chunkgrößenparameter sowie Remote Direct Memory Access (RDMA) Parameter, während bei LUSTRE Parameter für das Striping variiert wurden.

Bei der Betrachtung der Ergebnisse zeigten sich kein Effekt durch die Variation der Clientparameter. Allerdings sind durch Variation der Chunkgröße sowie der RDMA Parameter Unterschiede aufgefallen. Während bei der Chunkgröße je nach Anwendung unterschiedliche Größen die beste Performance zeigen, konnte bei der Variation der RDMA Parameter kein Performancegewinn gegenüber der Standardeinstellung festgestellt werden. Für LUSTRE zeigen kleine Blockgrößen beim Striping sowie eine größere Anzahl an Striping Targets die beste Performance in den eingesetzten Tests.

Stichwörter: HPC, Dateisysteme, BeeGFS, BeeOND, LUSTRE, Benchmarking

Abstract

To ensure optimal performance for applications in the field of high-performance computing, a fast, parallel file system is important. Therefore, the effects of file system parameters on the performance of applications were examined in the context of this work. Mainly the parameters of the BeeGFS file system, which was created for the duration of an application execution, were varied. Tests were also performed on a central LUSTRE file system. Five benchmarks were used to run tests with different parameter combination in order to test the performance of the file system for different use cases.

Client parameters, chunksize parameters as well as Remote Direct Memory Access (RDMA) parameters were considered for BeeGFS, while striping parameters were varied for LUSTRE.

When looking at the results, no effects could be observed by varying the client parameters. However, differences were noticed by varying the chunksize as well as the RDMA parameters. While the best chunksize setting for the application performance varies per application, no performance gain could be observed when varying the RDMA parameters compared to the default setting. For LUSTRE, small block sizes for striping and a larger number of striping targets show the best performance in the tests used.

Keywords: HPC, File Systems, BeeGFS, BeeOND, LUSTRE, Benchmarking

Contents

List of Figures xi

List of Tables xi

1 Introduction 1

2 Fundamentals 3
2.1 HPC system architecture . . . 3
2.2 Parallel file systems . . . 4
2.3 Burst Buffers . . . 6
2.4 Parameters on HPC storage . . . 7
2.5 Benchmarking on HPC . . . 8
2.6 Related works . . . 9
2.6.1 Tune applications for I/O performance . . . 10
2.6.2 Libraries for optimized I/O access . . . 10

3 Methods 13
3.1 System configuration . . . 13
3.2 Benchmarks chosen . . . 13
3.2.1 NPB BT-IO . . . 14
3.2.2 IO500 . . . 15
3.2.3 ZFS . . . 16
3.2.4 CIAO . . . 17
3.2.5 HiBench . . . 17
3.3 Disregarded benchmarks . . . 18
3.3.1 OpenFOAM . . . 18
3.3.2 BigDataBench . . . 19
3.4 Benchmarking workflow . . . 19
3.4.1 Benchmark series . . . 20
3.4.2 LUSTRE Benchmarking . . . 20
3.5 Evaluated parameters . . . 21
3.5.1 BeeGFS parameters . . . 22
3.5.2 LUSTRE parameters . . . 23

4 Results 25
4.1 Results on BeeOND . . . 25
4.1.1 Blocksize . . . 25
4.1.2 Client parameters . . . 27

4.1.3 RDMA parameters . . . 27
4.2 RAM usage for BeeOND . . . 28
4.3 Results on LUSTRE . . . 28

5 Discussion 35
5.1 RAM Usage of BeeOND . . . 35
5.2 Parameter tuning on BeeGFS . . . 35
5.2.1 Chunksize . . . 35
5.2.2 Client parameters . . . 37
5.2.3 RDMA parameters . . . 38
5.2.4 Caveats . . . 39
5.3 Stripe size tuning on LUSTRE . . . 42

6 Conclusion and future works 45

Bibliography 47

List of Figures

4.1 BeeOND chunksize executions . . . 26
4.2 BeeOND client parameter executions . . . 30
4.3 RDMA Parameter executions . . . 31
4.4 Worst-case RAM usage for BeeOND . . . 32
4.5 BT-IO on LUSTRE . . . 33

List of Tables

3.1 RDMA parameter configs . . . 23

1 Introduction

High Performance Computing (HPC) is an area of computer science where complex computational problems are solved by the use of supercomputers. Modern supercomputers utilize an architecture where many computers, so called nodes, are connected into one logical unit. These are interconnected with a fast, low latency network connection and have access to a common file system, with file system sizes up to hundrets of Petabytes. There are different domains that make use of supercomputers, including weather forecasting, fluid dynamics simulations, quantum computer simulations and many more. Usually supercomputers, also called clusters, are shared multi-user systems, with multiple users being able to use different parts of the same supercomputer for their scientific research.

Traditionally, HPC is using parallel file systems to read and write data. With rising application demands for storage capacities or throughput and increasing computational performances of recent CPUs and accellerators, traditional parallel file systems like LUSTRE [1] or IBM GPFS [2] increasingly hit their limit and become the bottleneck of application performance.

To reduce the impact of parallel file system performance on the application performance, a new storage layer of burst buffers was introduced. This is a storage layer optimized for high throughput, typically build on SSDs and designed to be available throughout the duration of a job.

This thesis focuses primarily on benchmarking BeeOND, an on-demand version of the BeeGFS file system, used as burst buffer. With BeeOND, it is possible to spawn transient file systems for a short duration, like a singular job, on an HPC system. When the job is finished, the file system can be destroyed afterwards. BeeOND can be spawned directly on the nodes that a job is using for computation. This file system is not suited for long-term storage, as all of storaged data is deleted as soon as the job is finished. Instead, it can be used as a buffer to store short-term data, with the benefit that this file system is exclusively available to that job and does not need to share bandwidth with co-located jobs.

Benchmarking BeeOND has two advantages over benchmarking traditional parallel file systems in an HPC environment:

1. **Consistency:** Due to exclusive access to the file system, it produces much more consistent result than using a shared file system. While results may still fluctuate due to node characteristics or network bandwidth based on node allocations, but they do not fluctuate nearly as much as on a shared resource.

2. **Rapid parameter changing:** Many file system parameters are bound to the lifetime of a file system. Traditional parallel file systems are heavy to deploy

and therefore changing parameters is a heavy operation. But as the lifetime of a BeeOND file system is limited to a single job execution, parameters can be changed here for each job execution.

All benchmarks are executed on the RWTH CLAIX-2018 cluster, utilizing the already existing infrastructure without modifications in the hardware. Therefore all performed steps are tailored to the environment present on this cluster.

This thesis will start to lay out some fundamentals, which are helpful to understand the thesis, in Chapter 2. In Chapter 3 the methods for benchmarking are shown. Next, in Chapter 4 the results of these benchmarks are described and in Chapter 5 discussed. Finally, in Chapter 6 the key essences of this work are concluded and possible future directions based on this work are shown.

2 Fundamentals

The following sections explain key concepts that are implicitly used throughout this thesis. These key concepts reach beyond the general scope of computer science and are rather specific to High Performance Computing (HPC).

Additionally in Section 2.6 papers and related works that have already investigated I/O optimization in an HPC context are presented.

2.1 HPC system architecture

HPC is an area in computer science area that has already been present since the 1960s, with the CDC 6600 being considered the first supercomputer, deployed in 1965 [3]. In computer science history, there has always been a demand for fast systems with more memory than an average system, especially for compute-intensive applications like scientific simulations.

While in the early days of HPC it was sufficient to have a single machine with a single, fast processor, physical limits prevent the scaling of single core machines, such that HPC first shifted to multi-core and later to multi-node architectures, where a supercomputer is built up from many nodes, all interconnected through a network.

As opposed to typical computers for interactive use, HPC systems are mainly used for batch processing, which means that compute jobs are submitted and scheduled for later execution when enough resources are free to process that compute job.

Todays HPC systems' core element are the compute nodes, which are the nodes used for batch processing. They are usually equipped with these crucial main components, which determine their performance:

1. **Processor(s):** The processor executes the computational work. Today, multi processor architectures are common, having multiple processors in the same node.

2. **Memory:** The memory is used to store application data. Many applications have a certain demand to the available memory, such that they can only be processed on certain nodes which have the required amount of memory available.

3. **Interconnect:** For communication between nodes, an interconnect system is used. A low latency is required for many applications, meaning that the time between sending and receiving a message on the other end should be below 1 microsecond. Therefore ethernet is not a suitable option here. Instead,

HPC usually utilizes the InfiniBand [4] network or Omni-Path [5]. The network topology also plays an important role in the overall system performance. However, this topic will not be covered in this thesis due to a different focus.

4. **Accellerator (optional):** While CPUs are very versatile when it comes to their ability for each core to perform independent operations, other architectures like GPUs can perform a significantly higher amount of operations per seconds with the same power usage, given the application structure supports this. Therefore some HPC systems are either fully or partly equipped with accellerators, mainly GPUs, which perform certain tasks with a higher efficiency than CPUs.

In addition to the compute nodes, there are login nodes, which are used to compile software and submit jobs. They are equipped similarly to a compute node, but accessible interactively.

The chosen operating system for both the login nodes and compute nodes on todays fastest 500 supercomputers are different Linux distributions, according to the Top500 list [6]. A scheduler is needed in order to schedule the requested jobs to the compute nodes. It has different tasks like making sure that these compute nodes are in a healthy state when executing jobs there, trying to reduce idle times on compute nodes and creating proper isolation for the jobs to ensure that they cannot access the resources from other jobs. For these (and some more) tasks CLAIX-2018 is using the slurm scheduler, which also has plugin support to allow for customized job startup behavior.

Dedicated storage is required on HPC systems to persistently store the datasets needed for applications. The architecture is described in the following section.

2.2 Parallel file systems

HPC is known to work with data sets up to many terabytes in size, requiring file systems with a corresponding size and good I/O performance to efficiently load and save these data sets [7].

Another requirement for HPC is a central file system across all cluster nodes, as HPC jobs should not be bound to nodes where the data is stored.

As single storage servers cannot satisfy the performance or capacity requirements for HPC environments, even with RAID configurations, HPC environments utilize parallel file systems.

Parallel file systems are software-based file systems which span single file systems across multiple servers while ensuring that all client systems have the same, consistent view on that file systems. Similar to a RAID system, parallel file systems split files into blocks of a certain size and distribute these blocks across different disks on different machines. Typical parallel file systems are designed to scale up to thousands of disks managed with a single file system.

Due to their distributed architecture, parallel file systems are more complex than local file systems, as they need to serve thousands of clients while making sure that files are not getting corrupted, but still support parallel access to the same file from clients in a performant way.

As all relevant HPC machines use Linux as their operating system, parallel file systems need to support the POSIX design which Linux uses as an interface for I/O, so called POSIX-I/O semantics. While most of the POSIX-I/O semantics are around the file system API, features they need to support (like permissions) and the possibility to be mounted, POSIX-I/O semantics also make assumptions about the integrity of files [8]. The most important assumption is the read-after-write consistency assumption, which means that any read of a file that takes place after a write is guaranteed to have the most recent version of that file available, even when the write and read take place on different machines [9]. This read-after-write consistency is not only required for the content of a file, but also for its metadata, and requires complex locking mechanisms on the parallel file system servers.

Parallel file systems are optimized for having a high throughput for read and write operations on big files, with multiple Gigabytes per seconds. Concurrent access of many small files is less performant for parallel file systems, as small files have a similar overhead regarding maintaining metadata and read-after-write consistency as big files, but require this overhead at a much higher frequency.

Traditional parallel file systems used in HPC are IBM GPFS [2], under a proprietary license, and LUSTRE [1], open source licensed with GPL2. Both file systems have huge production deployments:

- **GPFS:** Summit at Oak Ridge National Laboratory, 250PB of storage with 2.5TB of throughput [10]
- **LUSTRE:** Fugaku at Riken Center for Computational Science, 150PB of storage with 1.5TB of throughput, using the vendored LUSTRE release FEFS by Fujitsu [11]

Both LUSTRE and GPFS utilize distinct metadata and storage servers, where metadata servers manage the integrity of file metadata and storage servers manage the actual content of the files. Metadata includes information about where the content of a file is stored, last access time, filename, inode number, owner of a file, permissions and more.

BeeGFS

Another parallel file system used in High Performance Computing environments is BeeGFS, originally called FhGFS and developed at the Fraunhofer Institute for industrial mathematics [12]. This file system is the primary file system used for this thesis. Both terms BeeGFS and BeeOND will be used in the following. BeeGFS is the name of the file system and BeeOND (BeeGFS On Demand) is a provisioning method for BeeGFS, where a BeeGFS file system will be created and destructed via

script dynamically within the matter of seconds, typically to be used as a transient file system. Other parallel file systems do not provide an option to easily create and destroy a file system for the duration of a job.

As many file system parameters are bound to the lifetime of a file system, on demand provisioning allows to vary these parameters for different benchmark executions.

2.3 Burst Buffers

While traditional parallel file systems are sufficient for a large amount of applications, some applications with high I/O demands are bottlenecked by the performance of the file system [13]. As the file system is a shared resource on an HPC cluster, these applications do not only spend a long time in I/O, they also slow down other jobs with I/O demands running in parallel on the HPC cluster.

To migitate this issue and support a higher I/O bandwidth, the use of a high-bandwidth and relatively low-capacity storage tier, so-called burst buffers emerged in recent years. Technically, these burst buffers work similarly to parallel file systems, but are optimized for faster access and less redundancy, as they do not replace a traditional parallel file system, but are used as a caching layer for these file systems, either transparently or by user-initiated explicit copying of data between the different storage layers. Burst buffers are composed of fast storage mediums, typically NVMe SSDs, to satisfy the I/O demand of applications.

For final deployment of a burst buffer, there are 3 common approaches used for placing the burst buffers [14]:

1. **Node-local Burst Buffers:** The storage is placed within each compute node. This allows for low latency access without the need of additional hardware besides the compute nodes.

2. **Grouped Burst Buffers:** There are Burst buffer groups, placed network-topologically near the compute nodes on dedicated hardware. This allowes for easier sharing of data between compute nodes than with node-local Burst Buffers.

3. **Global Burst Buffers:** Burst Buffers are deployed similarly to traditional parallel File Systems as a global file system. This approach enables applications to maximize their I/O bandwidth for a short amount of time, as theoretically all of the storage devices in the Burst Buffer deployment are available for I/O, but as it utilizes a global network, this approach might create additional congestion, especially if the network is also used for the global parallel file system or application communication.

For this thesis, Node-local Burst Buffers are of interest, as this is the approach utilized at the RWTH cluster.

2.4 Parameters on HPC storage

The I/O performance of parallel file systems or burst buffers on HPC systems depends on the composition of possible parameters, both depending on the hardware and software used.

Hardware parameters

The underlying hardware is a crucial aspect for the file system performance. Hardware parameters start on a low level at the disk characteristics, with characteristics like sequential disk performance, random access operations per seconds, the supported access protocols or their size. The class of storage device makes a big difference here. Typical classes of storage devices used today are spinning disks (attached via SATA or SAS) or flash disks (attached via SATA, SAS or PCI over NVMe). While spinning disks are cheaper per Gigabyte and can provide good sequential I/O performance, their random I/O performance is generally low, as it involves physical seeking. In contrast to that there are flash disks with both a better sequential I/O performance and much better random access performance, but at a higher price tag. Depending on the actual disk, they might support the NVMe protocol, allowing for direct I/O without CPU interaction required, given the software stack supports it.

Going up a layer, there is the architecture of the storage and metadata servers, on parallel file systems typically split, on Burst Buffers these might be the same servers. Other than choosing the correct disks, here it is only important that the hardware which is chosen is capable of processing enough data to saturate the disks performance, which is a task typically performed by the storage vendor.

The next aspect that can parametrized on the hardware is the network. The network setup between storage servers and compute nodes determines the latency and the maximum available bandwidth on a per-node basis as well as globally. Different network topologies are common in recent supercomputers: On the one hand there are statically routed network topologies like a fat-tree, on the other hand there are also dynamically routed topologies like the dragonfly network or a variation thereof.

Software parameters

While the parameters regarding hardware configuration are crucial for the overall I/O performance, these parameters cannot easily and without significant cost be changed on an existing deployment. For existing HPC clusters, software parameters can be tuned in order to get an improved I/O performance.

We will not focus on the available software parameters on the application side here. A subset of those are covered in Section 2.6.

The most important tuning parameter is the selection of the file system that is used for I/O. As HPC environments utilize Linux as their operating systems, they use parallel file systems with POSIX semantics, such that they can be mounted as a

network file system and provide certain guarantees on consistency. These guarantees impose additional overhead on the file system and therefore reduce the possible performance.

Different types of file systems or file system configurations can be chosen:

- **Fully POSIX compliant FS:** These file systems are fully POSIX compliant and guarantee all of the POSIX assumptions about a file system. They can be mounted on the compute nodes similarly to a local file system
- **Relaxed POSIX compliant FS:** These file systems appear to the user similar as the fully POSIX compliant file systems, but relax some assumptions that may lead to unexpected results for certain access patterns. Applications should opt-in into these relaxations, as they need to make sure to not use these access patterns in order to ensure data integrity.
- **Object Storage:** In object storage the datasets are stored as Key-Value pairs, where a key is an identifier to create, read, update or delete the value of a dataset. As opposed to POSIX file systems, object storages are accessed via an API, typically via HTTP. The two most important APIs are the S3 API (created by Amazon) and the Swift API (created by OpenStack).

  Object storages opts for an eventual consistency model. This means that after updating an object it is not guaranteed that subsequent accesses to that object will directly provide the updated version of that object. It is only guaranteed that the updated version will be available "eventually" at a later point of time, reducing the number of locking mechanisms the storage is required to implement.

When opting for a certain file system type, the actual implementation of that file system type is another parameter for the file system performance.

(Relaxed) POSIX compliant storages have additional options that can affect the performance. In this area, network parameters can be tuned, for example buffer sizes for Remote Direct Memory Access (RDMA), which is a common low-latency networking protocol used in HPC. The chunksizes and striping counts can also be tuned: Internally, parallel file systems split files into multiple blocks of a certain size and store them to multiple disks. The amount of disks a single file is stored to as well as the size of a chunk can be tuned on parallel file systems, for some parallel file systems even on a per-folder or per-file basis.

2.5 Benchmarking on HPC

Benchmarking is a method to assess the performance of applications or the system configuration.

Unfortunately, benchmarking storage on HPC is not as easy as doing classical computing benchmarks where the CPU or GPU performance is assessed on a single

computer. Benchmarking on HPC is usually done with the submission of different jobs for different benchmark runs, resulting in benchmark runs on different machines. When benchmarking storages on HPC, several factors can influence the final resulting performance measured:

- **Disk degradation:** Storage mediums degrade over time, which is true for HDDs as well as SSDs, such that their performance gets worse before the end of their lifespan. As parallel file systems have a huge number of different disks, used disks for a benchmark execution and therefore their degradation vary from run to run.
- **Seek timing on spinning disks:** As spinning disks need to do physical seeks for initial access of a block, these seeks may take shorter or longer based on how much the disk needs to spin to seek to the correct position. This timing can vary from measurement to measurement. Fortunately, on SSDs no physical seek is required and random access is much more consistent.
- **Networking path and congestion:** As compute nodes are allocated for each run, they might have different network pathes for each execution, both to each other as well as to the file system. Additionally, in case they do not have a direct connection, their network path might be congested by co-located jobs.
- **File system congestion:** When using a shared file system, co-located jobs might be using that file system, reducing the available resources for the running benchmark.

For all of the above reasons, benchmarks measuring HPC I/O tend to have a higher variance than typical computing benchmarks. Therefore multiple iterations of benchmark runs with same parameters are required.

An additional limitation on HPC is that benchmarking is using compute and storage resources without actually yielding scientific results and therefore blocking parts of the HPC system for scientific applications. Benchmarks should only be carried out as much as necessary to gain insights on their results. While already in theory reducing the number of executed benchmarks contributes to a responsible use of resources, in practice there is the additional limitation of quota on the compute time and storage resources for an account, so benchmarks need to be planned in a way to stay within that quota.

2.6 Related works

I/O-performance is important for many HPC applications. As application performance is one of the crucial aspects in HPC, optimizing I/O access is part of that optimization process for these applications.

Much work has already gone into this topic, some of which is presented here.

2.6.1 Tune applications for I/O performance

To get a good application performance, HPC applications that perform heavy I/O need to adapt their I/O strategy in order to lower the time spent in I/O and therefore lower the overall runtime of the application.

An example for this approach is an I/O optimization approach for the CIAO application, presented in the case study by Göbbert et al. [15].

This application has a simulation step and a visualization step which needs to work on the simulated data. By introducing a change in the application and do the visualization in-situ rather than performing it as a post-processing step after running the simulation, all relevant data is already in the memory and an I/O step, that was formerly required for visualization can now be skipped.

Another example is presented for the GRAPES application, a numerical weather prediction application, where optimization has been done by Zou et al. [16].

In this case study two parallel I/O strategies were implemented in addition to the already available sequential I/O strategy. One implementation is based on MPI-IO and the other implementation is based on the ADIOS library, which are presented in Section 2.6.2. Specific configurations were fine-tuned to maximize the I/O performance on the Tianhe-1A as well as the Sunway Bluelight systems they are executed on. With help of these I/O libraries, the percentage of runtime spent in I/O could be reduced from over 50% with serial I/O to just 9% with parallel I/O on 2048 cores when utilizing the ADIOS library. This lowers the overall time spent in the application and therefore lowers the overall runtime of the application.

In order to not waste computing time, many relevant HPC applications are tuned within an optimization cycle. Naturally, when I/O becomes a bottleneck, it will also become a part of that optimization cycle.

While this approach tries to improve application I/O performance similarly to this thesis, it differs in the fact that it modifies the application. Therefore, this approach is not universally appliable to any application with low I/O performance.

2.6.2 Libraries for optimized I/O access

Traditional parallel file systems tend to have a good throughput, but they have a relatively low metadata performance [17]. This is why applications tend to perform better if they are working on a few rather big files instead of a huge number of small files.

In order to combine the flexibility of storing multiple datasets separately and on the other hand only touching a few files, there are file specifications with according interfaces like HDF5 [18], MPI-IO [19] or NetCDF [20] which can store multiple datasets in a single file and therefore circumvent the classical bottleneck.

A different approach is used with the ADIOS [21] (ADaptable I/O System) software, with a second version ADIOS 2 [22] released in 2020, where no dedicated file format is required to interact with the library.

The interface of the ADIOS software is similar to the traditional file interfaces

like `fopen()` for C/C++ and `open()` for Python, aiming for an easy integration into existing applications, while using a config file to fine-tune parameters for optimizing I/O performance.

Another problem for parallel access to the same file is false sharing, where multiple processes are accessing the same block of a file system at the same time. In order to ensure integrity for the file, the file system needs to make sure that the only one process can access a specific part of the file at the same time, commonly implemented with a locking mechanism. This can result in a major performance penalty for the application.

Lower-level libraries like SIONlib [23] allow parallel access with multiple processes to single files while minimizing false sharing. Libraries for accessing HDF5, MPI-IO and NetCDF also make sure that false sharing is minimized.

These libraries are beneficial for a good I/O performance while keeping the application maintainable. Using a library differs from the approach used in this thesis as it adapt the application I/O stucture to the file system specifics, while the approach in this thesis tries to adapt the file system characteristics to the I/O structure of applications.

3 Methods

To obtain results on the impact of file system parameters to the performance of HPC applications, benchmarks are carried out. The following sections describe the conditions under which the benchmarks were executed. First in Section 3.1 the system configuration is described, as the underlying hardware is the basis for the benchmarks that are executed on it. Then in Section 3.2 the chosen benchmarks and the rationale behind that choice is presented, with Section 3.3 showing benchmarks that were considered additionally, but decided against. In Section 3.4 the workflow for carrying out benchmarks and gathering parameter details as well as results in a structured way is described. Finally, in Section 3.5 the parameters that have been varied throughout different benchmark executions are shown.

3.1 System configuration

All of the tests were performed on the HPC cluster CLAIX-2018 at RWTH Aachen University.

It consists of 1032 nodes each with 48 cores Intel Skylake (2 $\times$ Xeon Platinum 8160), 384 GB of memory and a SATA-SSD with a capacity of 480 GB. Nodes are interconnected with Intel Omni-Path 100G connections.

The local SSD is shared by the BeeGFS On Demand File system and the operating system, leaving about 400 GB per node for the BeeGFS file system. BeeGFS is storing internal files on a local XFS partition with a blocksize of 4 KiB.

3.2 Benchmarks chosen

As the effect of varying file system parameters should be evaluated, some tests need to be done in order to verify whether a parameter change has an effect and if so, how big that effect is. Benchmarks are the tool of choice here, as they provide metrics which quantify the impact of these parameter changes. The absolute values gathered from the benchmarks are not of interest here, as we only want to know whether a parameter change had an effect and we do not need to know what the best possible I/O performance on a benchmark is, possibly requiring code changes. Relative values compared to a default configuration are of a much higher importance here.

As the impact of these parameter changes for different scenarios should be tested, multiple benchmarks provide different insights into different I/O behaviors for ap-

plications. A well representative set of benchmarks should include many of the following requirements:

- **I/O dependency:** As we want to test the file system, a dependency of the tests to I/O configuration is a hard requirement in order to assess whether they perform better or worse with a given configuration.
- **HPC relevance:** Benchmarks should be relevant for usage in different HPC areas, as that is the realm we want to test and possibly optimize applications for.
- **Mix of application and microbenchmarks:** While synthetic microbenchmarks are best when it comes to stress-testing certain areas of the file system, application benchmarks are better in showing the real-world relevance of parameter tuning for file systems.
- **Dedicated I/O statistics:** To assess the impact of the file system changes to the application, an isolated view to the I/O portion (or even multiple I/O stages) of the application is beneficial, as it reduces the variance introduced by other factors.
- **Configurability:** To tailor benchmark executions to the CLAIX environment, benchmarks with parameters that determine the size of the workload are preferred. An absolute requirement is the configurability of the amount of nodes or processors used for the benchmark.

In the following subsections, the selected benchmarks for this thesis are described in detail.

3.2.1 NPB BT-IO

The BT-IO benchmark is part of the benchmark suite NAS Parallel Benchmarks (NPB) developed by NASA [24].

This benchmark is an I/O-intensive version of the Block Tri-diagonal solver application [25]. It has a bursty usage pattern, as after every five steps results are written to disk.

Requirement fullfillment

This benchmark tests a specific solver, which is a typical use case for HPC. Here an algorithm is tested, which is somewhat between a microbenchmark and a full-blown application benchmark, as it does not directly test the raw file system performance, but rather an algorithm that is executed within this benchmark. It does output dedicated I/O statistics, although only a single value which is the combined throughput of the benchmark. As for configurability, this benchmark has the disadvantage that it does not allow for arbitary core counts, but they need to be a square number, as

explained in the following. It has the option to be configured for different workload sizes though, which change the amount of I/O required.

Configuration used

Due to the nature of this benchmark, it has a limitation in that it only works for square MPI Thread counts. Therefore the number of threads for each execution is chosen by the maximal possible number of threads that fit within a job allocation:

$$t = \left\lfloor \sqrt{c \cdot n} \right\rfloor^2$$

Where c is the number of cores per node, n is the amount of nodes allocated for a benchmarking job and t is the final number of threads.

The Benchmark Size D is chosen for parameter tests with embarassingly parallel I/O, writing 136 GB per benchmark run. Within the I/O step of the embarassingly parallel version of this benchmark, each MPI process uses its own file for I/O usage.

3.2.2 IO500

The IO500 benchmark is a composite benchmark consisting of different microbenchmarks measuring the I/O throughput and metadata performance under various circumstances [26].

It consists mainly of the benchmarks IOR and mdtest, but it also includes a pfind-based test. IOR tests the storage bandwidth, while mdtest tests the metadata performance of a file system. Pfind is an additional metadata test, utilizing a parallel find implementation to search for a file. Both the IOR and mdtest are executed in different scenarios to accomodate best-case and worst-case IO performance. IOR tests the read and write performance in different stages, while mdtest tests the create, stat and delete performance in different stages.

For IOR, the easy scenario is a scenario where all processes write sequentially to their own file, which is a favorable scenario for sequential I/O. On the hard scenario all processes write the same file, with each processor writing 47008 Byte, creating misaligned I/O and possibly creating a false sharing scenario, depending on the file system implementation. The same procedure is repeated for the read scenario.

For mdtest, the easy scenario creates empty files within a balanced directory structure, while the hard scenario creates empty files within the same directory. After creating these files, additional stages are used for testing stat and delete operations for the same files.

Requirement fullfillment

This benchmark is a collection of microbenchmarks, which means that it has a relevance to HPC as it outputs raw values for I/O performance. While not being an HPC application itself, the relevance for this filed is given as these values can be taken as a reference for the maximum expected I/O performance in ones own

application. For each microbenchmark I/O metrics are given, such that these can easily be comprehended. The configurability of the IO500 benchmark is very good, as the amount of processors is freely configurable, as well as size of the workload for each step of the benchmark.

Configuration used

The IOR part of the IO500 benchmark is using MPI-I/O to read and write data to disk. This benchmark ran with 8 processors per node.

Due to instabilities in BeeGFS, the mdtest-easy phase files were lowered to 200000 files per process, while for the mdtest-hard phase the number of used files was lowered to 10000 per process. IOR easy is limited to 40 GB per process, resulting in 320 GB written per node on average to accomodate the local disk size of about 400 GB.

3.2.3 ZFS

The Zonal Flow Solver (ZFS) is a multiphysics simulation software that is used for solving problems as fluid-structure interaction, aeroacoustics, or fuel injection, combustion and flow problems in human respiration [27]. This application was used as a benchmark example in the RWTH JobMix for procurement of the CLAIX-2018 system.

The benchmark application is a simulation of aeroacoustics that is acting on a cartesian mesh. It relies on I/O for writing the results after each simulation step back to the disks.

The I/O pattern of the ZFS benchmark is primarily saving the final results of the simulation back to disk.

Requirement fullfillment

This benchmark is an application benchmark used in HPC, so the relevance in this area is given. It provides dedicated I/O statistics for the I/O, as fine-grained logs show the details on the I/O timing of the application. Unfortunately as this is a specific application, the configuration of the workload size not possible without domain knowledge for this application. The execution with an arbitary amount of processors is possible though, which is the reason this benchmark was still included.

Configuration used

The same test set that was already used in for the CLAIX-2018 procurement is used here. This is a test set where an acoustic perturbation equation is solved. Within this benchmark five files, each having a size of 20 GB, are created.

3.2.4 CIAO

The CIAO application is a multiphysics simulation software used in the area of combustion technology [28]. This application was also used as a benchmark example in the RWTH JobMix for procurement of the CLAIX-2018 system and executed here with the same test set.

Requirement fullfillment

Similarly to ZFS, this benchmark is also an application benchmark for an application actively used in HPC. It provides I/O statistics for each performed I/O step which can be used as metrics. Similarly to ZFS, also this application does not allow for easy adaption of the workload size without domain knowledge of the application. Also here the amount of processors can be arbitarily chosen, so this benchmark was also included.

Configuration used

Within the benchmark a “Droplet Injection” dataset is used. This is an I/O intensive workload with five steps of writing application checkpoints between iterations back to disk and a sixth step to write the final results to disk.

3.2.5 HiBench

To not only include traditional HPC workloads but also Big Data workloads, the HiBench suite [29] is used to benchmark file system parameters with the Big Data applications Apache Hadoop and Apache Spark [30], utilizing a Hadoop File System (HDFS) [31] as their storage layer. Big data software is used for large-scale statistical analysis and machine learning on that data, utilizing data sets that can become many Terabytes in size.

HiBench consists of multiple big data benchmarks for different workloads, including microbenchmarks, websearch benchmarks, machine learning benchmarks and a graph benchmark.

Requirement fullfillment

While historically most HPC applications are in the area of mathematical modelling and simulations, machine learning and other data science based workloads are becoming increasingly relevant for HPC, for which Big Data software can be utilized. Big Data workloads also have different I/O access patterns than simulation applications and give additional insights into the parameter effect onto these workloads. With a great variety of sub benchmarks included with HiBench, it does not only feature microbenchmarks, but also some benchmarks for real-world scenarios. Each sub benchmark includes its own execution time, which is a sufficient metric. HiBench offers multiple preconfigured workload sizes, but even has an option to fine

tune the workload size for individual benchmarks. Setting the amount of nodes or processors is possible, but works here a differently than in a typical HPC application, as a scheduler service is involved here and the amount of processors used is controlled by that scheduler and not by the execution of the benchmark.

Deployment

To obtain reproducible results, a new Hadoop/Spark cluster is initiated running on the allocated nodes of a job. One of those nodes is initiated as the master node which does not run any workloads, but only coordinates the workloads for the other nodes. For deployment, the magpie script is used [32].

HDFS is deployed on top of the BeeOND file system, comparable to a simple network-based file system. In general, this may not be the optimal strategy for HDFS deployment on a productional cluster, as HDFS benefits from data-locality which is hidden through the BeeOND layer. As we are mainly interested in relative and not absolute results, this is not an issue in this scenario, as all benchmark runs have HDFS deployed in the same way.

Configuration used

The profile “large” is used, as a compromise between workload and execution time. Some benchmarks were excluded due to incompatibilities to the installed version of Scala, having a large execution time or failing during execution. All HiBench runs are using one node more than comparable other benchmarks as a master node is required, which only delegates tasks and does not perform the actual workload computations.

3.3 Disregarded benchmarks

While further benchmarks were considered for testing within the context of this thesis, not all criterias as described in Section 3.2 could be satisfied. Therefore they were not used to execute tests with.

3.3.1 OpenFOAM

The OpenFOAM application is a simulation software in the area of computational fluid dynamics. This application needs to perform a considerable amount of I/O, as simulation inputs need to be read from disk and simulation outputs need to be written back to disk.

While this application is widely used within HPC clusters, using it as a benchmark would mean to run dedicated scenarios where I/O is performed. Without domain knowledge in the area of computational fluid dynamics it is hard to find proper scenarios for this application, as many examples are only possible to process in specific versions of OpenFOAM. Adapting the amount of cores that process this

example also requires domain knowledge of this application and the scenario that should be simulated.

Additionally, OpenFOAM does not output dedicated I/O statistics, such that the only metric possible to process without adapting the source code would be the full execution time of the application. This introduces an additional variance, possibly hiding the effect of I/O to this application behind the cumulated variance of the whole application run.

For these reasons, OpenFOAM was not used as a benchmarking tool within the scope of this thesis.

3.3.2 BigDataBench

Another big data benchmark that was considered is BigDataBench [33], similarly to HiBench also a benchmark that analyzes the performance of Big Data workflows.

Setting up Big Data stacks on an HPC system is difficult, as it requires various services running in order to execute the required workloads. For these services specific versions are needed, which make the deployment even harder, as for example the magpie script used for HiBench does not support all versions of Hadoop and Spark. BigDataBench is even more complex to setup than HiBench, as it also requires a relational Database as a service for running benchmarks, in addition to the Big Data software stack using, among others, HDFS, Hadoop and Spark.

Because Big Data workloads are not primarily HPC workloads, there is already another Big Data benchmark included for this thesis and the difficulty to setup BigDataBench, it was not used as a benchmarking tool within the scope of this thesis.

3.4 Benchmarking workflow

All benchmarks require submission via slurm to be executed, as that is the job scheduler on CLAIX-2018. A spank plugin is responsible for modifying config values in the BeeGFS config files. These config values are given as a slurm parameter in the job script.

To submit benchmarks to slurm, a custom python script is used that sets the relevant parameters for execution via jinja2 template. Within this python script, file system parameters can be easily adjusted.

As seen in Listing 1, all parameters are dumped to the job output within the submission script. This parameter dump is used to easily comprehend which benchmark with which parameters was used in a job submission for evaluation. The json format was chosen as it is a machine-readable format with simple structure and a good tooling support, while still being human-readable.

```json
{
    "tasks": 32,
    "nodes": 4,
    "job_type": "beeond",
    "benchmark": "io500",
    "darshan": false,
    "beeond_settings": {
        "helperd": {
            "tuneNumWorkers": "2"
        },
        "mgmtd": {
            "tuneMetaSpaceLowLimit": "10G"
        },
        "storage": {
            "tuneFileReadAheadSize": "0m"
        },
        "meta": {
            "tuneNumWorkers": "0"
        },
        "client": {
            "tuneFileCacheType": "buffered"
        },
        "general": {
            "chunksize": "128K"
        }
    },
    "series": "chunksize128K-4nodes-v2"
}
```

Listing 1: Example parameter dump for benchmark execution (abbreviated, more BeeGFS parameter settings in original output)

3.4.1 Benchmark series

To gather results for evaluation, an additional python script is submitting a "benchmark series", which is a group of all benchmarks executed with a certain parameter set, submitted multiple times to identify and reduce variance.

In Listing 1 there is a series name under the key "series". Each series is submitted with a name configured, such that they can easily be grouped within the evaluation.

3.4.2 LUSTRE Benchmarking

While the main focus of this thesis is benchmarking applications on BeeGFS, some tests on the central LUSTRE were performed. While the LUSTRE file system cannot be recreated on each benchmark run due to operational constraints, LUSTRE supports changing striping parameters on a per-folder basis.

As LUSTRE is a global and shared file system, it has different constraints opposed to BeeOND benchmark runs:

- **Shared resource:** As the LUSTRE filesystem is a shared resource, co-located jobs might introduce a higher variance than runs on the BeeOND file system with exclusive access.
- **Quota limitations:** Quota on the global LUSTRE file system was limited to 1TB of data and 50000 inodes for the benchmarking account. Therefore only a limited amount of benchmarks is still viable for execution on LUSTRE

A similar approach as the benchmarking series in Section 3.4.1 is used here. To always have the same load on the file system for all jobs, it is preferred that all jobs, which should be compared, are starting at the same time, such that all benchmarks have roughly the same load during benchmark execution.

The job submission is done in waves, such that not all repetitions of a series are scheduled at the same time, but rather one repetition of multiple benchmarks are scheduled at the same time. When all of these benchmarks are finished, the next repetition is scheduled.

Due to the quota limitations, especially the inode limitation, it was not possible to execute each benchmark in a sensible configuration. BT-IO benchmark was tested on LUSTRE, as it on the one hand still runs well with the limited quota and on the other hand promises significant results as we will see in Section 4.1.2. As the used configuration of this benchmark requires to store 136 GB to disk, with a quota of 1000 GB a maximum of $\lfloor \frac{1000GB}{136GB} \rfloor = 7$ executions can run in parallel.

3.5 Evaluated parameters

In order to obtain information about specific tested parameters, a selection was made which parameters to include in testing.

Only parameters that possibly influence the performance of the file system in a productional deployment were considered. Many settings are not relevant to the performance, but are rather used to get the file system working in the environment they are deployed to. Additionally, there are debugging settings, that may cause a performance degradation when enabled and should stay disabled on a productional deployment for this reason.

Types of parameters that were not considered include these:

- Port configurations
- File locations
- Log level
- Timeout settings
- Authentication and authorization settings

On the CLAIX-2018 machine the installed BeeGFS version is BeeGFS 7.1.5. All parameters considered were already available with that version of BeeGFS, further parameters that were introduced in newer versions of BeeGFS were not considered.

3.5.1 BeeGFS parameters

The tested BeeGFS parameters can be categorized within the following categories:

1. **Chunksize:** The internal size of a file chunk, as stored by BeeGFS for striping. These were tested in the range between 4 KiB and 16 MiB, with each step multiplying the previous value by 4. Due to results discussed in Section 5.2.1, the additional chunksizes 32 KiB and 128 KiB were considered. Note that the chunksize is independent of the internal blocksize on the underlying file system where BeeGFS stores data.

2. **Client parameters:** Different tuning parameters that influence consistency guarantees on BeeGFS. These parameters are tunable in a config file. Considered boolean parameters were [34]:

   - **tuneUseGlobalAppendLocks:** Controls whether files opened in append mode should be protected by locks on the local machine only (=false) or globally on the servers (=true).
   - **tuneCoherentBuffers:** Enables or disables coherence between the buffers used by `tuneFileCacheType=buffered` and the page cache.
   - **tuneRemoteFSync:** Controls whether `fsync()` syscalls from a user application should only be executed on the client to transfer data from the client cache to server cache (=false); or also on the servers to flush the servers cached file data to the disks (=true).
   - **tuneUseGlobalFileLocks:** Controls whether application advisory file locks via `flock()` and `fcntl()` should be checked for conflicts on the local machine only (=false) or globally on the servers (=true)

   Disabling these parameters might on the one hand improve performance, but applications might expect a more consistent behavior, potentially leading to applications misbehaving within their I/O.

   Another test was executed with all parameters disabled, which would be the configuration that is expected to have the best performance, but most potential for application misbehavior.

3. **RDMA parameters:** Remote Direct Memory Access (RDMA) is a technology used for data transfer without CPU intervention, using Intel Omni-Path on the CLAIX-2018 cluster.

   For RDMA, the number of buffers and size per buffer per connection is configurable with the parameters `connRDMABufNum` and `connRDMABufSize`.

Four different configurations of RDMA parameters were tested, as listed in Table 3.1. As each RDMA buffer needs to reserve memory, the configs have an increasing memory demand and can assess to which extend it makes sense to trade memory for performance. The required buffer per connection is as follows:

$$\text{connRDMABufNum} \cdot \text{connRDMABufSize} \cdot 2$$

Note that there are multiple types of connections for communication in different directions, most notably from client nodes to storage nodes and from client nodes to metadata nodes. Multiple connections between two nodes are possible for increased throughput. In the performed tests a maximum of 12 connections per pair of nodes was possible.

Table 3.1: RDMA parameter configs

Config	`connRDMABufNum`	`connRDMABufSize`	RAM per connection
rdma_very_low	10	256	5 KiB
rdma_low	35	1024	70 KiB
default	70	8192	1.1 MiB
rdma_high	120	16384	3.75 MiB

Furthermore all of the nodes allocated for a job are used both as metadata servers as well as storage servers. While it makes sense to use all available nodes as storage servers to get the maximum possible throughput and disk capacity, tweaking the amount of metadata servers would be an interesting parameter to tune. The amount of metadata servers may impact the stability of the file system. Additional metadata servers may increase the performance of the file system due to increased parallelism, but it may also be possible that additional metadata servers decrease the file system performance, as more currency control is required, where the overhead might be bigger than the performance gain due to increased parallelism. However this parameter could not be considered due to operational constraints.

3.5.2 LUSTRE parameters

For LUSTRE, only striping settings were considered for testing.

4 different stripecounts were tested, namely 1, 2, 4 and 6 stripes per file. For each stripecount test, 6 different striping sizes in the range between 64 KiB and 64 MiB were tested, with each step multiplying the previous value by 4. LUSTRE requires a minimum striping size of 64 KiB, therefore no lower value could be tested.

As the LUSTRE tests were executed on a long-running, shared file system, testable parameters were limited to parameters that can be adjusted at runtime without interrupting the availability of the file system. Therefore, no other parameters could be tested here.

4 Results

In this chapter, selected results from benchmark execution are presented and visualized with error bars. In Section 4.1 results for different BeeOND executions are presented. Later in Section 4.3 the results for LUSTRE executions are shown.

These results will be discussed later on in Chapter 5.

4.1 Results on BeeOND

In the following subsections results of the BeeOND benchmarks in the different parameter categories discussed. The different categories that were benchmarked are presented in the same order as in Section 3.5.1.

4.1.1 Blocksize

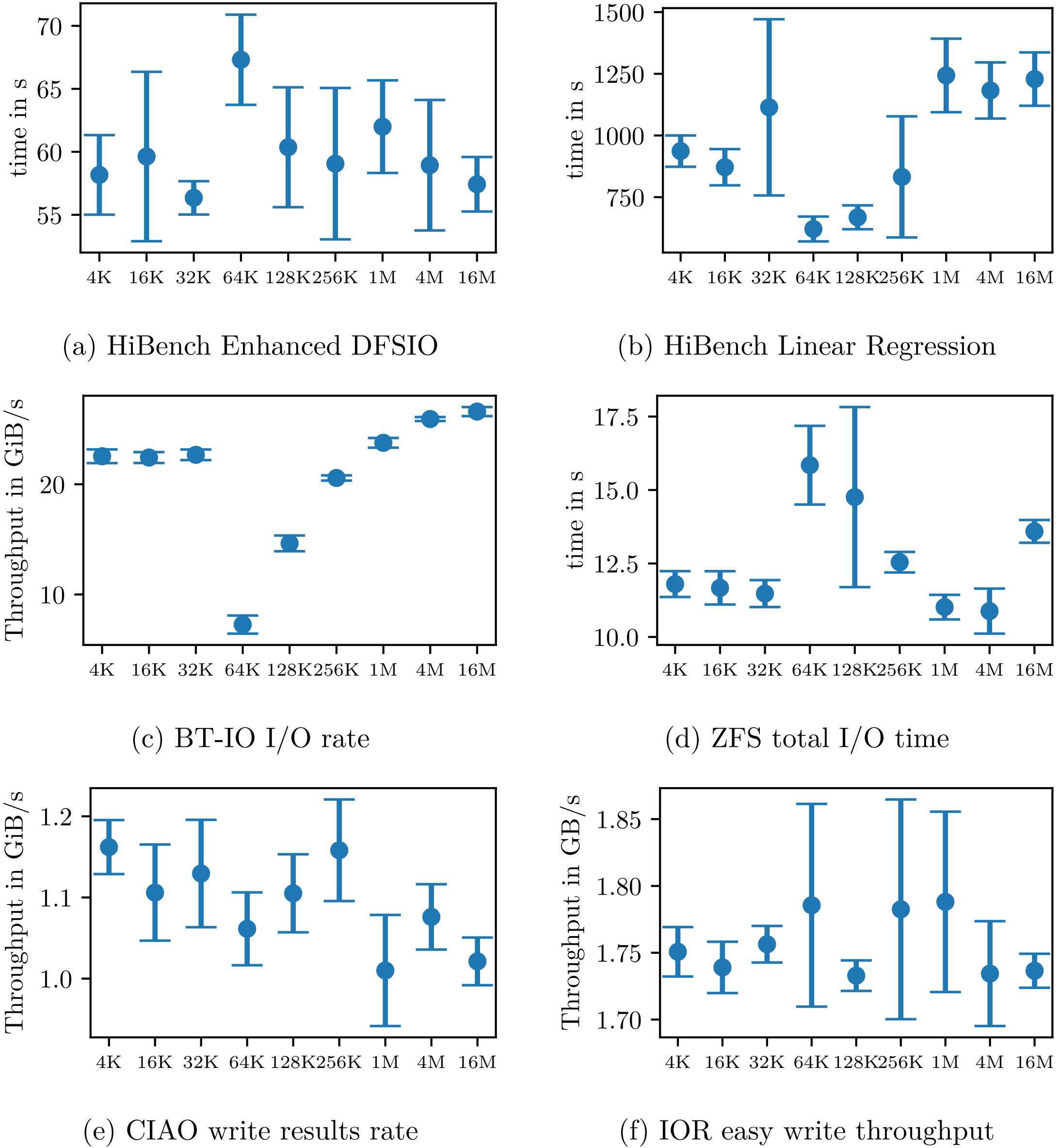
Benchmark series runs for chunksize parameters were performed as described in Section 3.4.1. Similarly to LUSTRE, BeeGFS allows limiting to how many disks a single file is striped. However this parameter was not adopted, as it would limit the available bandwidth and storage for workloads working with big files.

In Figure 4.1 results of selected metrics are shown. The benchmarks were executed on 4 compute nodes and were run 5 times. Error bars are assuming a normal distribution and provide a confidence interval of 95%. The metrics shown are only a limited selection of all available metrics, as these metrics show most significant results for their respective benchmark.

For the HiBench benchmarks in Figure 4.1a and Figure 4.1b, results are quite different. While in DFSIO most parameters produce comparable results, except for the chunksize of 64 KiB that performs worse here, the linear regression workload seems to have the best performance at 64 KiB, taking around 600 seconds while both chunksizes up to 32 KiB and from 1 MiB take significantly longer and need up to double the amount of time for linear regression.

The BT-IO execution in Figure 4.1c shows results with a low variance. Here the chunksizes 4 KiB, 16 KiB and 32 KiB show a throughput of about 22 GiB/s. A chunksize of 64 KiB shows a very significant drop with only about 7.5 GiB/s throughput for I/O operations. The bigger the chunksize get, the better the performance gets for this benchmark, obtaining a maximum throughput of about 27 GiB/s for a chunksize of 16 MiB.

The ZFS execution in Figure 4.1d shows a similar pattern on the 64 KiB chunksize, also performing significantly worse than most other executions. The I/O per-



(a) HiBench Enhanced DFSIO

(b) HiBench Linear Regression

(c) BT-IO I/O rate

(d) ZFS total I/O time

(e) CIAO write results rate

(f) IOR easy write throughput

Figure 4.1: BeeOND chunksize executions. The data points in each graph represent the result for a chunksize that was tested with their error margins. On the x axis the used chunksize is shown.

formance for the maximum chunksize of 16 MiB seems to lower here, taking about 13.6 seconds while most chunksize only need about 12 seconds of I/O time.

Both the CIAO execution in Figure 4.1e and the IOR execution in Figure 4.1f show a high variance in their results, with the latter of them not showing any significant results and the former having irregular patterns for executed benchmarks, showing a slight trend of better results for lower chunksizes.

4.1.2 Client parameters

Benchmark series runs for client parameters were performed as described in Section 3.4.1.

In Figure 4.2 results of selected metrics are shown. The benchmarks were executed on 4 compute nodes and were run 5 times. Error bars are assuming a normal distribution and provide a confidence interval of 95%. The metrics shown are only a limited selection of all available metrics, because they reflect the general direction of the other metrics.

Across diffent benchmarks, all parameters show a moderate variance, and do not indicate that changing any of these parameters makes a significant difference. In the ZFS run on Figure 4.2d the `noCoherentBuffers` series as well as the `noGlobalFileLocks` series has a much larger variance which can be explained by an outlier measurement with an I/O time of around 17 seconds for both.

Note that that the `allOff` series is a combined series with the `tuneRemoteFSync`, `notuneCoherentBuffers` and `tuneUseGlobalFileLocks` parameter disabled.

4.1.3 RDMA parameters

Benchmark series runs for RDMA parameters were performed as described in Section 3.4.1. Results of five selected metrics from all tested benchmarks with different amount of RAM consumption for RDMA necessary were performed according to Figure 4.3. The benchmarks were executed on 4 compute nodes and were run 5 times. Error bars are assuming a normal distribution and provide a confidence interval of 95%.

Across all selected metrics the `default` configuration produces best results, while the configuration with higher RAM requirements has no significant difference for three of those benchmarks, only in Figure 4.3a and Figure 4.3c this configuration performs sligthly worse than the `default` configuration.

Across all selected metrics both the `veryLow` as well as the `low` configuration perform significantly worse than the `default` configuration. For the mdtest, HiBench and CIAO benchmarks in Figure 4.3a, Figure 4.3b and Figure 4.3e the `low` configuration performs about 10% worse than the `default` configuration with the `veryLow` having similar values, except for the CIAO benchmark where the `veryLow` config is about 15% worse, but with a higher error margin.

For the ZFS benchmark in Figure 4.3d the `veryLow` and `low` configurations need about 20% longer than the `default` configuration.

For the BT-IO test in Figure 4.3c the `veryLow` and `low` configurations underperform compared to the `default` configuration, with the `default` configuration having a throughput that is about 2.2 times higher than the other two configurations.

4.2 RAM usage for BeeOND

RAM is a valuable resource on HPC systems, as many applications need many Gigabytes of RAM, some applications even take as much RAM as provided in order to maximize their caching.

To assess RAM usage required for running BeeOND in different scenarios, a python script monitored the RAM usage for the different BeeOND processes over time.

In Figure 4.4 the maximum RAM usage is shown, from the benchmark execution where the peak of cumulative RAM usage of the client, helper, metadata and management process was highest. This graph is from a HiBench benchmarking run from the veryHigh RDMA config as described in Section 4.1.3, using 5 nodes.

While the RAM usage for the management and helper process are both relatively constant and low with about 23 MiB and 6.4 MiB RAM usage, the RAM requirements for the Metadata process as well as the Storage process is rising over time and both require a higher amount of RAM for operation. The final RAM consumption for the Storage service is at 340 MiB, while the Metadata service requires 875 MiB of RAM for operation, totalling to a combined RAM usage for all BeeOND processes on a single node of 1244 MiB.

Over the first 30 minutes, the RAM that is required for the metadata and storage services is rising, but around that time mark the RAM usage stays on a constant level.

Note that the graph chosen here represents the worst-case scenario, with all other runs requiring less RAM for the BeeOND services.

4.3 Results on LUSTRE

BT-IO tests with LUSTRE as backing file system were performed as described in Section 3.4.2. Results of four executions with 1, 2, 4 or 6 stripes are visible in Figure 4.5. The benchmarks were executed on 4 compute nodes and were run 5 times. Error bars are assuming a normal distribution and provide a confidence interval of 95%. It should be noted that the different graphs in that figure are not directly comparable to each other, as they were executed at different point of times with a different workload from co-located jobs by other users.

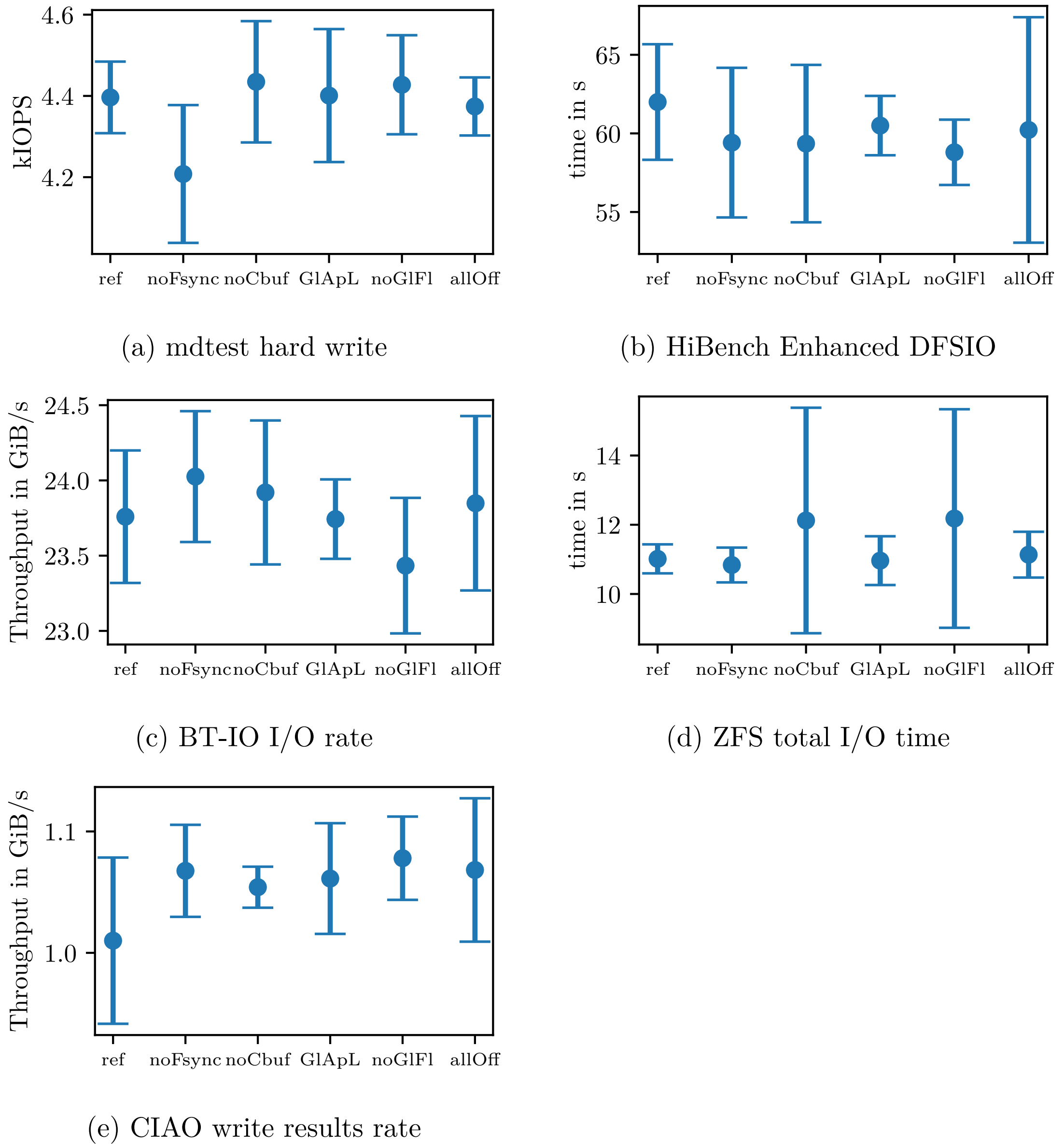
While in Figure 4.5a, where a stripecount of 1 is used, all chunksizes have a comparable throughput with about 17 GB/s, there are differences for the other stripecounts.

For all benchmark runs with a stripecount of 2, 4 and 6 it can generally be seen that the throughput of the benchmark is better for smaller chunksizes. For the chunksizes 16 MiB and 64 MiB we can see that across all stripecounts their performance stays at about 17 GB/s, comparable to all results for stripecount 1.

In Figure 4.5b, where a stripecount of 2 is used, the chunksizes 64 KiB and 256 KiB show improved throughput over the other chunksizes, where 64 KiB shows the

biggest improvement.

In both Figure 4.5c and in Figure 4.5d, with a chunksize of 4 and 6, there are roughly 3 plateaus, with 64 KiB and 256 KiB on the first plateau, 1 MiB and 4 MiB on the second plateau and 16 MiB and 64 MiB on the third plateau, achieving worse performance with each plateau, while the 1/4 MiB plateau has a better throughput with a stripecount of 6 than with a stripecount of 4.



(a) mdtest hard write

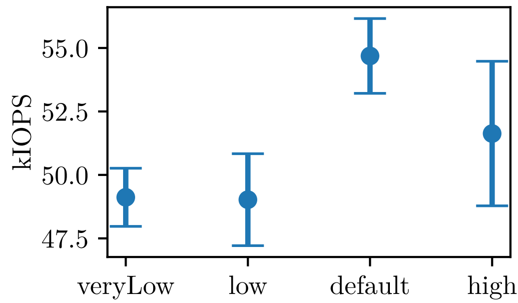
(b) HiBench Enhanced DFSIO

(c) BT-IO I/O rate

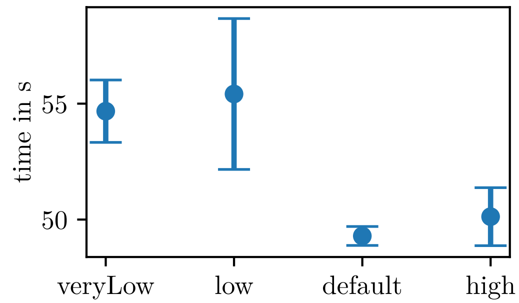
(d) ZFS total I/O time

(e) CIAO write results rate

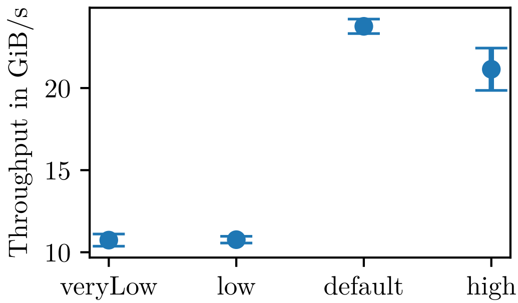
Figure 4.2: BeeOND client parameter executions. The 6 data points in each graph represent the results for a certain parameter set with their error margins. Following parameters were used: `ref`:reference run with no parameters changed, `noFsync`:`tuneRemoteFSync` set to false, `noCbuf`: `tuneCoherentBuffers` set to false, `GlApL`: `tuneUseGlobalAppendLocks` set to true, `noGlFl`: `tuneUseGlobalFileLocks` set to false, `allOff`: All of the above parameters set to false



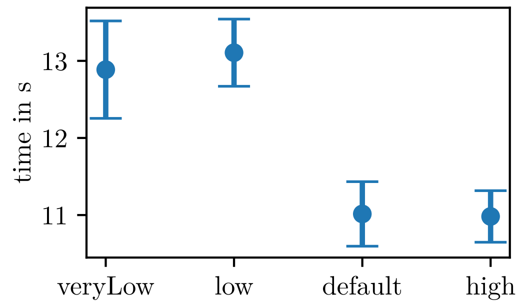
(a) mdtest combined results



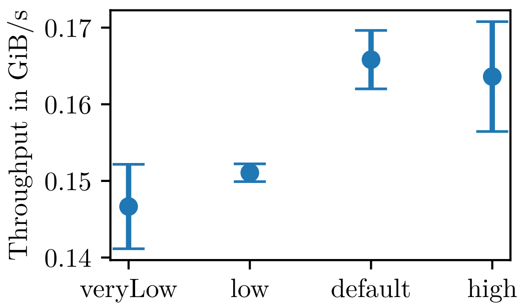
(b) HiBench Scala Spark Sorting



(c) BT-IO I/O rate



(d) ZFS total I/O time



(e) CIAO write rate for first execution

Figure 4.3: BeeOND RDMA parameter executions. The 4 data points in each graph represent the result for a certain RDMA parameter configuration with their error margings, as listed in Table 3.1.

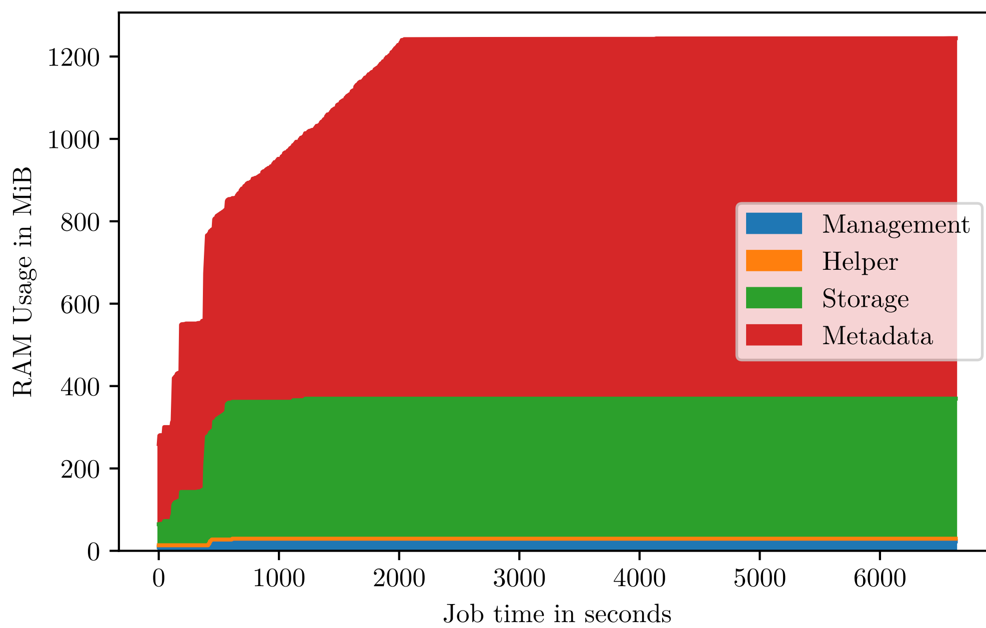
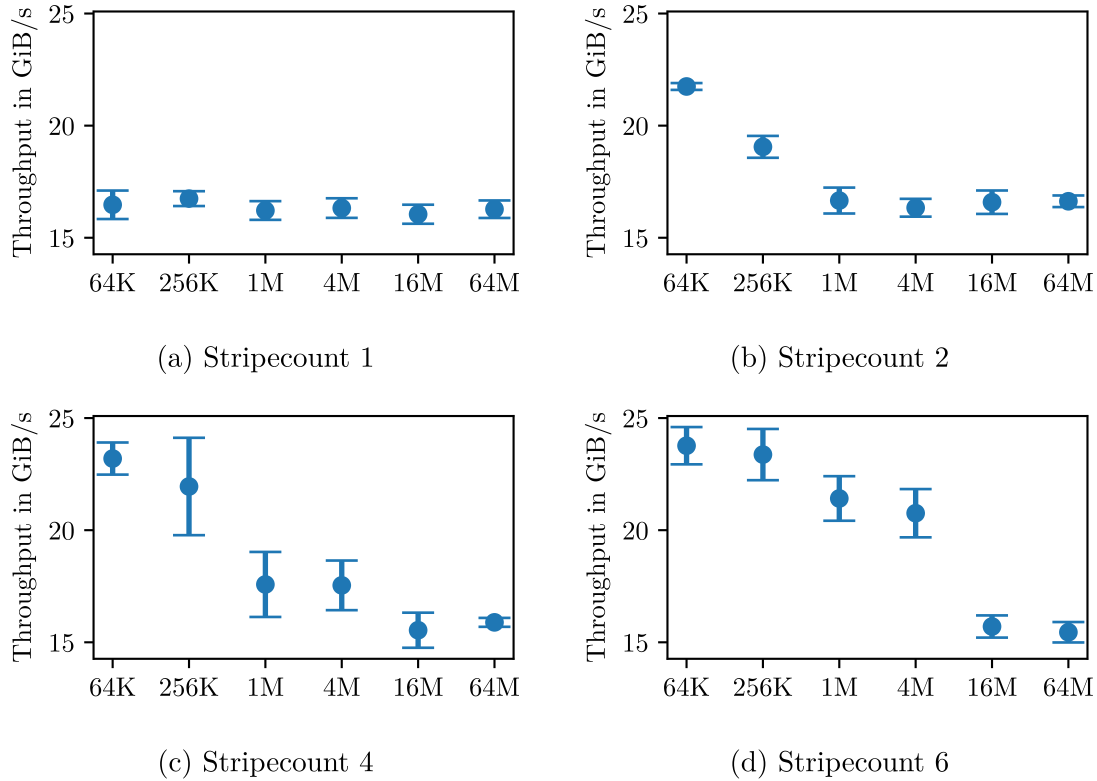


Figure 4.4: Worst-case RAM usage for BeeOND



(a) Stripecount 1

(b) Stripecount 2

(c) Stripecount 4

(d) Stripecount 6

Figure 4.5: BT-IO Benchmark execution on LUSTRE. For each graph, the stripecount was chosen differently, denoting the amount of disks used for each single file. The data points show the result for the BT-IO execution, with error margin included. The x axis shows the LUSTRE striping size.

5 Discussion

In this chapter the results from the previous chapter are evaluated and possible explanations for the root cause of varying performance are given.

At the end of an evaluation of a certain parameter set, a recommendation for tuning this parameter set is given. This recommendation can be used as guideline for tuning ones own application for optimized I/O.

5.1 RAM Usage of BeeOND

When creating a parallel File System on compute nodes during job duration, applications need most of the available RAM, possibly creating conflicts with the RAM requirements of the file system. As outlined in Section 4.2, BeeOND had a maximum RAM usage of 1244 MiB in the worst case of all performed benchmarks. Todays Supercomputers typically have hundreds gigabytes of RAM per compute node, leaving the reserved memory for the BeeOND file system at less than 1% and therefore do not increase the memory requirements considerably.

Note that the maximum amount of nodes tested in this scenario were 5. Additional nodes may require a larger memory footprint, especially for RDMA connections, as more point-to-point communications are established when the amount of storage servers as well as metadata servers is increased.

Additionally the underlying file system where BeeGFS data is stored might have its own caching mechanism. Typical underlying file systems like ext4, XFS or ZFS are deeply integrated into the linux kernel and use much of otherwise free RAM for their internal caching. BeeOND I/O performance might benefit from a lower application memory footprint, as more of the data BeeGFS uses would still be cached.

5.2 Parameter tuning on BeeGFS

In Section 4.1 multiple parameters for BeeGFS were adjusted to test their impact on the different benchmarks executed. In the following subsections the impact of these parameters will be evaluated.

5.2.1 Chunksize

In Section 4.1.1 different BeeGFS chunksizes were tested. Testing was initially conducted with the chunksizes between 4 KiB and 16 MiB, where each step multiplies

the previous value by 4.

Note the caveats described in Section 5.2.4, which, among others, explain why the throughput for BT-IO is above the theoretically available throughput.

Performance at 64 KiB What can be noticed is that a chunksize of 64 KiB performs poorly across different benchmarks, including HiBench Enhanced DFSIO in Figure 4.1a, ZFS total I/O time in Figure 4.1d and especially for the BT-IO benchmark in Figure 4.1c. For all of the above executions, both the 16 KiB as well as the 256 KiB performed better than the execution with a chunksize of 64 KiB.

Because of that result, additionally the chunksizes 32 KiB and 128 KiB were considered, as they are half and double the chunksize of 64 KiB. With these additional chunksizes it is visible that the performance for a chunksize of 64 KiB is indeed very low across three different, independent benchmarks.

Possible explanations for these could be that there is a bug in the BeeGFS source code that causes this performance degradation. Another possibility is that some algorithmic parts are changed at the threshold of 64 KiB, for example another buffering strategy is used with a chunksize $\geq$ 64 KiB which is known to work better for bigger chunksizes.

Unfortunately it is out of scope of this work to investigate the root cause for this behavior, as the issue may lay in the underlying file system, somewhere in BeeGFS or in hardware characteristics. Deep understanding of the BeeGFS internals as well as general file system internals in Linux are needed in order to debug this issue.

BT-IO performance For BT-IO, the performance shows a clear trend: While the throughput for chunksizes between 4 KiB and 32 KiB are rather constant at about 22 GB/s, the throughput is very low at 64 KiB with around 7.6 GB/s, but then getting better as the chunksize increases. With this value almost doubling with a chunksize of 128 KiB and an I/O rate of around 14.9 GB/s, it is likely that at a chunksize of 64 KiB the amount of processed chunks is the limiting factor, as in both cases around 125,000 chunks are processed per second.

With an increasing chunksize, the throughput gets increasingly limited by a maximum bandwidth which seems to be at around 28 GB/s, while a chunksize of 1 MiB seems to be the first chunksize with higher throughput than 22 GB/s, which is observed between 4 KiB and 32 KiB.

One explanation for this behavior could be that bigger chunksizes are more efficient in doing I/O and have less overhead and therefore a better overall throughput. As BT-IO is doing many short, bursty I/O operations, another explanation for this behavior could be that the I/O is finished from an application point of view as soon as the last block has been cached by BeeGFS and does not need to be actually written to disk. Bigger chunksizes have an advantage here, as a bigger amount of data could be cached here.

Other results CIAO as well as Linear regression seem to benefit from smaller chunksizes, as seen in Figure 4.1e as well as Figure 4.1b, with CIAO performing worse with a chunksize $\geq$ 1 MiB, while the Linear Regression workflow has the best performance with a chunksize between 64 KiB and 128 KiB, which is the chunksize that other workloads are struggling with.

Tuning recommendation

The chunksize can make a huge difference in the application performance and should be tuned. As this tuning is available on a per-directory and even on a per-file basis, no special setup is needed for doing user-level tuning here, even when BeeGFS is not used as an on-demand file system.

As for the actual values that are beneficial, testing should be tailored to the application, as different applications benefit from different chunksizes. Critical I/O sections should be benchmarked seperately as opposed to measuring the whole application runtime to gain better insights on the I/O performance of an application.

Executing a few tests starting at a chunksize of 4 KiB and then multiplying each subsequent test by 4, up to a chunksize of 16 MiB, gives a good overview on the performance gains or losses. These results show which chunksizes are beneficial for an application.

As a general rule of thumbs, if an application uses many small files, a small chunksize will be beneficial for that application, while applications that only work with a few files that are many Gigabytes or even Terabytes in size bigger chunksizes are beneficial.

5.2.2 Client parameters

In Section 4.1.2 results for modified client parameters can be seen. Across all different benchmarks, these parameters do not seem to have any effect on the results of those benchmarks. This is even true for the `allOff` config. Here all client parameters were set to false, potentially saving some I/O overhead and determining the best case for performance. The fact that no effect can be seen could mean that either the overhead associated with these operations is quite small, or that the scenarios where these parameters actually make a difference are not triggered by the application. Some parameters are specific to certain syscalls, for example `tuneUseGlobalFileLocks` has an influence on the `flock()` and `fnctl()` syscalls. When applications do not use these syscalls, either directly or indirectly via libraries, no performance difference is expected here.

While this on the one hand says that tuning of client parameters may not make sense for HPC applications, it is important to note that none of the applications behaved badly with these tunings enabled.

Given that `tuneCoherentBuffers` was set to `false` for one execution, this result was uncertain. While the expectation would be that coherent buffers might increase the performance of an application, disabling it might lead to inconsistencies if the

application is not aware of that behavior and actively preventing to run into scenarios where coherent buffers would be needed. This is equally true for the benchmark with the `tuneUseGlobalFileLocks`, as well as `tuneUseGlobalAppendLocks`, as both parameters influence whether locks are placed globally or only locally on a node, potentially circumventing locking mechanisms which applications may rely on.

In Section 5.2.4 some caveats on interpreting these results are shown.

Tuning recommendation

The tested client parameters did not seem to have an influence on the application performance or correctness. Therefore sticking to the default configuration should be sufficient for many applications when BeeGFS is used in a Burst Buffer scenario.

With mirroring turned on, client parameters may play a more important role on I/O performance, where tuning might be beneficial.

5.2.3 RDMA parameters

Remote Direct Memory Access (RDMA) is used as a technology for communication between BeeGFS clients and the storage server as well as the metadata server. As opposed to TCP/IP communication via ethernet, communication via RDMA is typically done on a separate device and does not need a CPU interrupt as soon as the corresponding buffer is filled.

Changing the RDMA parameters has an influence on the performance of various benchmarks. As shown in Section 4.3, 4 different configurations with different RAM usage were tested, with varying RAM usage numbers according to Table 3.1.

Again, the BT-IO benchmark is influenced by this setting, having more than double the throughput on the default configuration as opposed to the `veryLow` and `low` configuration, with about 24 GiB/s vs. 10.5 GiB/s.

This throughput difference is likely explained by having a buffer that is too low for having a consistent stream of I/O and the buffer is faster filled than the time required for a roundtrip to the storage server. The throughput on the `high` configuration is also worse than the default configuration, even though the RAM requirement is higher here. The amount of buffers is increased here, as well as the size of a single buffer. It is likely that increasing the amount of buffers does not induce any performance penalty in case the RAM required for these buffers is available, as these additional buffers can still be left unused in case they are not used. As already explained in Section 5.2.1, the I/O in this benchmark has a bursty profile. Increasing the buffer size for RDMA connections may lead to an increased latency for connections, which leads to a performance penalty for I/O transmissions of a small size, which is less relevant for I/O transmissions of bigger sizes.

Similar effects like with BT-IO can be observed in the mdtest combined results with the mdtest benchmark in Figure 4.3a, although the performance advantage for the default configuration is only at about 10% compared to the `veryLow` and `low` configuration.

Furthermore the other three shown benchmark metrics in Figure 4.3 show decreased performance with the `veryLow` and `low` config, although no significant difference can be found here between `default` and `high` configuration.

As these RDMA configurations primarily influence the communication between processes and mainly depend on the RDMA device, it is likely that a global setting exists here which works best for the specific RDMA device on a cluster.

Tuning recommendation

Tests have shown that the default configuration for RDMA parameters shows a good performance across a diverse set of benchmarks, so the default choices of 70 buffers with a size of 8 KiB per RDMA connection seem like a good choice to start with, at least with the Omni-Path devices present on the CLAIX-2018 cluster. Higher throughput RDMA devices may benefit from a bigger buffer size, but that would need to be tested on the according cluster. Cluster admins should therefore test different RDMA parameters on their cluster to obtain a performant default setting.

In case there is some RAM available which is not required by the job, the amount of buffers could be increased. The RAM usage for BeeOND RDMA buffers is capped at around:

$$\texttt{connRDMABufNum} \cdot \texttt{connRDMABufSize} \cdot 2 \cdot \texttt{connMaxInternodeNum} \cdot \texttt{nNodes}$$

This calculation needs to be done seperately for the storage and metadata server. In case values for both are the same, the maximum RAM usage reserved for RDMA is doubled.

5.2.4 Caveats

While above results are suggesting which parameters might have an impact on the application performance and which might not, in reality there are some caveats on interpreting those results. Some of them are explained in the following:

No significant results ≠ no effect Most of the executed benchmarks produce results with an error margin of at least 5%, mostly due to the nature of file systems. Some tuning options may actually have an effect that vanishes within the error margin. Without a completely isolated environment, where the exact same disks on exactly the same compute nodes are used and no network congestion is present, it is hard to lower the error margin. Even with such an isolated environment there might still be some variance in results due to the general flakiness in benchmarking file systems [35].

Selection bias for benchmark metrics In Chapter 4 only a small subset of all benchmark metrics is shown, which is expecially true for HiBench which provides metrics for 25 sub-benchmarks. As presenting all of those benchmark runs would

clutter this thesis with many insignificant informations, only a subset of interesting sub-benchmarks or metrics were selected to be shown. This selection might lead to the impression that many of the executed benchmarks provide significant results while in reality many sub-benchmarks or metrics without significant results were stripped from the work.

Low accumulated bandwidth With a maximum accumulated bandwidth of around 2-2.5 GB/s for 4-5 nodes, this bandwidth is not even on par with a single, modern NVMe SSD. This is a low bandwidth for HPC applications. In this given environment many parameter changes do not show any significant effect. However, with a higher bandwidth significant effects might show up with a higher load on the storage server, where the storage server might hit a throughput wall depending on the parameter configuration.

Unfortunately, no system with better storage devices was available for testing, leaving this hypothesis untested.

No mirroring configured As BeeGFS is only used in an on-demand way with BeeOND within this thesis, the impact of mirroring was not tested. Some parameters might have a different impact on the I/O performance with mirroring configured.

Configuring BeeGFS parameters while mirroring is enabled might additionally change the correctness behavior of applications I/O, as these parameters could influence the internal replication management of BeeGFS. Especially parameters configured in Section 5.2.2 could have a different impact on the I/O correctness with mirroring configured.

BT-IO above theoretical bandwidth BT-IO benchmark runs on BeeGFS report a bandwidth above the maximum possible bandwidth of four SATA ports, exceeding that bandwidth by a factor of up to 13.5 with 27 GB/s. Therefore results of the BT-IO benchmark should be treated with caution.

There are possible explanations for this behavior:

1. **Caching effects:** Results may not be written to disk, but only cached in RAM. As indicated in Section 5.1 the RAM usage of BeeGFS is rather low and not able to cache 136 GB, but the data may be cached on the XFS layer.

2. **Asynchronous I/O per process:** As the version of the BT-IO benchmark with embarassingly parallel I/O is used, different processes may be doing I/O at different points of time. The end result would then be summed up (or a calculation in a similar fashion). With 136 GB of data written to disk and a write bandwidth somewhat below 2 GB/s, a total I/O time of about 80 seconds is needed, which would be in a realistic range.

In case the explanation is the former, this benchmark would rather test the caching instead of the actual time to write to disk. That would be OK when used in a burst

```
ERROR: read(7, 0x7c9000, 2097152) returned EOF prematurely, (aiori-POSIX.c:725)
```

Listing 2: IO500 failure in `ior-easy-read` phase

buffer scenario, where an unexpected ram failure would lead to a failed job anyways. Then the caching would be tested, which is a reasonable aspect of a file system to test for.

In case the latter is the explanation for the observed behavior, this would also be fine, as all executions of the BT-IO benchmark would have the same behavior and the relative difference is of relevance here.

The source code shows that I/O is not synchronized in the embarassingly parallel version and I/O is not necessarily done completely in parallel. After calculating the I/O rate for each individual process, the cumulated I/O rate is the sum of these individual I/O rates. Caching behavior on the file system layer could still result in an additional increase of bandwidth though.

IO500 crashes On the default configuration IO500 ran into a bunch of crashes. Some crashes were related to filling up the file system with no space left for write operations, which could be resolved by capping the maximum write to disk.

However, even after capping the maximum amount of storage written to disk, IO500 still crashed in that modified default configuration with different error messages and at different stages, including `ior-hard-write`, `ior-easy-read`, `mdtest-easy-delete` and `mdtest-hard-read`, when executed with 4 or more threads per node. The file system did not provide any apparent reason for the failure, neither was the maximum amount of storage or inodes used ($> 90\%$ of available inodes for BeeGFS were unused), nor were there any log messages indicating a file system failure.

A sample error in the `ior-easy-read` phase is shown in Listing 2, with POSIX I/O used instead of MPI-IO. This error shows a premature EOF, which could either mean that a the read operation has failed or that the file, which got created during a previous `ior-easy-write` phase, has been corrupted, indicating the possibility of a silent data corruption.

Tests were executed with 8 threads per node and testing loads were further lowered to make IO500 executions pass reliably. On the one hand, these configuration changes lower the confidence to the correctness of the results provided by IO500, as lower loads might lead to a higher cache-hit rate than intended by the IO500 benchmark. On the other hand, the frequent crashes raise concerns in the desired stability of the BeeGFS file systems. While IO500 is a stress test to the file system purposely testing the limits of a file system, a well-behaving file system should only become slow in case resources become a bottleneck and not fail, or even worse, silently discard I/O-calls while storage and inodes are available.

5.3 Stripe size tuning on LUSTRE

For LUSTRE, tests were executed where different striping sizes were tested, with results described in Section 4.3. Also the amount of disks that a file was striped to was varied. The LUSTRE striping size determines the size of a stripe of a single file, before the next stripe is placed to another disk. The stripecount on the other hand determines to how many disks a single file is striped to, where in this test up to 6 disks were tested.

For a stripecount of 1 no significant differences in thoughput for BT-IO could be found, with all striping sizes having a throughput of about 16 GiB/s. This result is not surprising, given that all stripes are placed on the same disk.

For the other stripecounts all striping sizes $\geq$ 16 MiB do not show an advantage over the results obtained with a stripecount of 1. However, when decreasing the striping size, the throughput increases, with a higher maximum for higher stripecounts, up to a maximum of about 24 GiB/s with a stripecount of 6 and a striping size of 64 KiB.

The fact that increasing the stripecounts increases the throughput is expected, given that more disks are available for striping data to them. However it is somewhat surprising that low striping also contribute to a better throughput. There are two contradicting hypotheses that would explain one or the other direction:

1. **Smaller striping sizes = higher overhead:** As the striping sizes get smaller, the management overhead per stripe would stay the same. Therefore a lower throughput would be expected as the amount of stripes that needs to be managed needs to be increased.

2. **Smaller striping sizes = better balancing:** As the striping sizes get smaller, different stripes are better balanced between different disks. This would mean that for a single file all participating disks can continuously do I/O for that particular file, while bigger striping sizes would mean that there are more idle times for these participating disks while other disks are doing I/O.

As already mentioned and visible in Figure 4.5, the second theory can explain the behavior and shows that smaller striping sizes have a better balancing, while the additional overhead associated with the first theory seems to be negligible in this case.

As LUSTRE is a shared resource, using very low striping sizes might lead to a high load on the LUSTRE servers which may degrade the performance for other users and in extreme cases even lead to instabilities on the LUSTRE file systems. As no access to the LUSTRE servers was possible, it could not be observed whether lower striping sizes increase the load on the file system significantly or to an alarming level.

Another thing to consider is that the used scenario was a bursty scenario for short-term storage and mainly the write performance was tested.

The read performance might differ here, especially when LUSTRE is not used as a short term storage, but as a longer term storage. In this case the cache hit rate for LUSTRE would likely be lower, as data was not accessed for a longer time.

Tuning recommendation

The striping pattern for LUSTRE should be chosen to stripe to a great amount of disks, even unlimited in case only a single big file is used, such that the LUSTRE file system utilize the combined throughput of many disks combined.

The striping size should be chosen as little as possible order to reduce idle times on disks when waiting for the next stripe and therefore having a good write performance. It remains to be evaluated whether small striping sizes of 64 KiB increase the load on a LUSTRE file system to a level where the file system gets unstable, in that case a bigger striping size should be chosen.

6 Conclusion and future works

In this thesis the effect of file system parameter changes on the performance of applications within the HPC context have been evaluated. The file system BeeGFS was analysed in a burst buffer context, where the file system exists for the duration of a compute job on the compute nodes. Furthermore, LUSTRE runtimes parameters were analysed in a separate set of tests.

For BeeGFS it has been shown that I/O intensive applications have a varying I/O performance depending on the chunksize, but the optimal chunksize depends on the application that is execuded. Modifying tuning parameters on the BeeGFS client process did not turn out to have a significant impact on the application performance. RDMA parameters have an impact on the I/O performance, but no configuration could improve the performance over the default setting here.

For LUSTRE, striping sizes as well as striping counts have been evaluated. These show that higher striping counts with lower striping sizes show best I/O performance.

As a follow up to this work, multiple directions are possible:

- **Re-evaluation on better and more isolated hardware:** All BeeGFS tests have been executed on a limited hardware, especially did they only utilize a single SATA-SSD per node, with network congestion still possible. An evaluation of the same tests on hardware with better and/or more storage devices per node might provide additional insight about limitations of a file system.

  Isolating this hardware would eliminate some jitter between different runs, especially on the network congestion and could therefore produce results with lower variance.

- **Develop workflow for (semi)-automatic chunksize adjustment:** Different applications benefit from different chunksizes. Therefore an easy workflow for finding the optimal chunksize for a given application would be a great opportunity for I/O optimization.

  This workflow could consist of a tool that executes an application multiple times with different chunksizes configured on the I/O folder, while monitoring the I/O of that application. Darshan [36] could be utilized for the purpose of I/O monitoring.

  An approach like this would have the advantage that other than configuring the main I/O directory of an application, no other adjustments to the application are necessary and tests could be executed with low efforts required.

Bibliography

[1] P. Schwan et al. Lustre: building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[2] F. Schmuck and R. Haskin. Gpfs: a shared-disk file system for large computing clusters. In *Conference on File and Storage Technologies (FAST 02)*, 2002.

[3] NCAR. Cdc 6600. `https://www2.cisl.ucar.edu/ncar-supercomputing-history/cdc6600`, 2022. [Online; accessed 2022-08-28].

[4] G. F. Pfister. An introduction to the infiniband architecture. *High performance mass storage and parallel I/O*, 42(617-632):102, 2001.

[5] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® omni-path architecture: enabling scalable, high performance fabrics. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, 2015. DOI: `10.1109/HOTI.2015.22`.

[6] TOP500.org. Operating system family / linux. `https://www.top500.org/statistics/details/osfam/1/`, 2022. [Online; accessed 2022-08-28].

[7] F. Z. Boito, E. C. Inacio, J. L. Bez, P. O. A. Navaux, M. A. R. Dantas, and Y. Denneulin. A checkpoint of research on parallel i/o for high-performance computing. *ACM Comput. Surv.*, 51(2), Mar. 2018. ISSN: 0360-0300. DOI: `10.1145/3152891`. URL: `https://doi.org/10.1145/3152891`.

[8] M. Vilayannur, S. Lang, R. Ross, R. Klundt, L. Ward, et al. Extending the POSIX I/O interface: a parallel file system perspective. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2008.

[9] G. Lockwood. What's so bad about posix i/o? `https://www.nextplatform.com/2017/09/11/whats-bad-posix-io/`, 2017. [Online; accessed 2022-02-16].

[10] S. Oral, S. S. Vazhkudai, F. Wang, C. Zimmer, C. Brumgard, J. Hanley, G. Markomanolis, R. Miller, D. Leverman, S. Atchley, and V. V. Larrea. End-to-end i/o portfolio for the summit supercomputing ecosystem. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, Denver, Colorado. Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: `10.1145/3295500.3356157`. URL: `https://doi.org/10.1145/3295500.3356157`.

[11] DDN. Ddn installs 30 sfa18k series storage hardware devices as second-tier storage for the supercomputer fugaku. https://www.ddn.com/press-releases/ddn-sfa18k-supercomputer-fugaku/, 2020. [Online; accessed 2022-02-16].

[12] J. Heichler. An introduction to beegfs. http://www.beegfs.de/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014.

[13] A. Ovsyannikov, M. Romanus, B. Van Straalen, G. H. Weber, and D. Trebotich. Scientific workflows at datawarp-speed: accelerated data-intensive science using nersc's burst buffer. In *2016 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 1–6, 2016. DOI: 10.1109/PDSW-DISCS.2016.005.

[14] H. Khetawat, C. Zimmer, F. Mueller, S. Atchley, S. S. Vazhkudai, and M. Mubarak. Evaluating burst buffer placement in hpc systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2019. DOI: 10.1109/CLUSTER.2019.8891051.

[15] J. H. Göbbert, M. Bode, and B. J. N. Wylie. Extreme-scale in situ visualization of turbulent flows on ibm blue gene/q juqueen. In M. Taufer, B. Mohr, and J. M. Kunkel, editors, *High Performance Computing*, pages 45–55, Cham. Springer International Publishing, 2016. ISBN: 978-3-319-46079-6.

[16] Y. Zou, W. Xue, and S. Liu. A case study of large-scale parallel i/o analysis and optimization for numerical weather prediction system. *Future Generation Computer Systems*, 37:378–389, 2014. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2013.12.039. URL: https://www.sciencedirect.com/science/article/pii/S0167739X14000053. Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications.

[17] S. R. Alam, H. N. El-Harake, K. Howard, N. Stringfellow, and F. Verzelloni. Parallel i/o and the metadata wall. In *Proceedings of the Sixth Workshop on Parallel Data Storage*, PDSW '11, 13–18, Seattle, Washington, USA. Association for Computing Machinery, 2011. ISBN: 9781450311038. DOI: 10.1145/2159352.2159356. URL: https://doi.org/10.1145/2159352.2159356.

[18] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, 36–47, Uppsala, Sweden. Association for Computing Machinery, 2011. ISBN: 9781450306140. DOI: 10.1145/1966895.1966900. URL: https://doi.org/10.1145/1966895.1966900.

[19] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong. *Overview of the mpi-io parallel i/o interface*. In *Input/Output in Parallel and Distributed Computer Systems*. R. Jain, J. Werth, and J. C. Browne, editors. Springer US, Boston, MA, 1996, pages 127–146. ISBN: 978-1-4613-1401-1. DOI: 10.1007/978-1-4613-1401-1_5. URL: https://doi.org/10.1007/978-1-4613-1401-1_5.

[20] R. Rew and G. Davis. Netcdf: an interface for scientific data access. *IEEE Computer Graphics and Applications*, 10(4):76–82, 1990. DOI: 10.1109/38.56302.

[21] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible io and integration for scientific codes through the adaptable io system (adios). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*, CLADE '08, 15–24, Boston, MA, USA. Association for Computing Machinery, 2008. ISBN: 9781605581569. DOI: 10.1145/1383529.1383533. URL: https://doi.org/10.1145/1383529.1383533.

[22] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky. Adios 2: the adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020. ISSN: 2352-7110. DOI: https://doi.org/10.1016/j.softx.2020.100561. URL: https://www.sciencedirect.com/science/article/pii/S2352711019302560.

[23] W. Frings, F. Wolf, and V. Petkov. Scalable massively parallel i/o to task-local files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, Portland, Oregon. Association for Computing Machinery, 2009. ISBN: 9781605587448. DOI: 10.1145/1654059.1654077. URL: https://doi.org/10.1145/1654059.1654077.

[24] D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The nas parallel benchmarks 2.0. *Encyclopedia of Parallel Computing*, Jan. 1996.

[25] P. Wong, R. VanderWijngaart, and B. Biegel. Nas parallel benchmarks i/o version 2.4, Dec. 2002.

[26] J Kunkel, J. Bent, J. Lofstead, and G. S. Markomanolis. Establishing the io-500 benchmark. *White Paper*, 2016.

[27] A. Lintermann, M. Meinke, and W. Schröder. Zonal Flow Solver (ZFS): a highly efficient multi-physics simulation framework. *International journal of computational fluid dynamics*:1–28, 2020. ISSN: 1029-0257. DOI: 10.1080/10618562.2020.1742328. URL: https://publications.rwth-aachen.de/record/788106.

[28] M. Bode, A. Y. Deshmukh, J. H. Göbbert, and H. Pitsch. CIAO: Multiphysics, multiscale Navier-Stokes solver for turbulent reacting flows in complex geometries. Technical report FZJ-JSC-IB-2016-01, 2016. URL: `https://publications.rwth-aachen.de/record/690878`.

[29] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: characterization of the mapreduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 41–51. IEEE, 2010.

[30] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[31] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. DOI: `10.1109/MSST.2010.5496972`.

[32] LLNL. Magpie source code. `https://github.com/LLNL/magpie`, 2022. [Online; accessed 2022-06-02].

[33] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: a big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–499, 2014. DOI: `10.1109/HPCA.2014.6835958`.

[34] BeeGFS. Beegfs default client config. `https://git.beegfs.io/pub/v7/-/blob/7.1.5/client_module/build/dist/etc/beegfs-client.conf`, 2018. [Online; accessed 2022-07-05].

[35] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: it *IS* rocket science. In *13th Workshop on Hot Topics in Operating Systems (HotOS XIII)*, Napa, CA. USENIX Association, May 2011. URL: `https://www.usenix.org/conference/hotosxiii/benchmarking-file-system-benchmarking-it-rocket-science`.

[36] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright. Modular hpc i/o characterization with darshan. In *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pages 9–17, 2016. DOI: `10.1109/ESPT.2016.006`.