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

Performance-Bewertung und Vergleich des NVMe-Speichers zu dem Lustre Dateisystem auf CLAIX

Performance evaluation and comparison of NVMe memory and Lustre File System on CLAIX

Bachelorthesis

Lukas Aldenhoff
Matrikelnummer: 333731

Aachen, den 15. März 2018

Communicated by Univ.-Prof. Matthias S. Müller

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter:	Prof. Dr. rer. nat. Uwe Naumann (*)
Betreuer:	Aamer Shah, M.Sc. (')

(') Chair for High Performance Computing, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Chair for Software and Tools for Computational Engineering, RWTH Aachen University

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 12. April 2018

Abstract

While today's HPC systems are advancing in concurrency and computational power as the number of nodes grows and CPU performance increases, the incorporated I/O subsystems improve only slowly with their main components advancing at a rate that cannot keep up with the growth of computational power.
HPC clusters incorporate distributed parallel file systems to handle all I/O operations that are requested on compute nodes. A main component of these parallel file systems are the Hard Drive Disks (HDD). The technology of HDDs has improved over time, but the rate cannot compare to the development of CPUs. Furthermore I/O behavior of modern HPC applications deviates in I/O intensity, resulting in an overall very bursty I/O workload. As a consequence there are many occasions where the I/O subsystem of a modern large compute cluster is under high pressure.
Although HDDs could be replaced by newer and more advanced technology like Solid State Drives (SSD), there are two concerns with this solution: SSDs are expensive and HPC users demand for large amounts of storage space, forcing most HPC system owners to use the much cheaper HDDs instead. Another concern is the complex structure of parallel file systems. Factors like the latency and achievable bandwidth on parallel file systems are not only depending on the storage device, but also on other components like the network connection and the I/O subsystems servers.
Therefore deploying SSD devices is neither feasible in terms of its cost, nor would it guarantee the desired performance gain, as other components of the parallel file system could become bottlenecks. This problem is even more apparent in the case of metadata performance, which heavily relies on factors like latency. With these developments we are in search for ways to assist the parallel file system with its weaknesses while still taking full advantage of its strengths.
This work puts forward a possible way to supplement a parallel file system with an on-demand Non-Volatile Memory Express (NVMe) SSD storage on compute nodes. For this the parallel file system BeeGFS is deployed on the compute nodes with the NVMe SSDs as storage devices. In this way the NVMe SSDs are usable as a small but coherent secondary parallel file system.
This thesis selects, simulates and tests realistic use-cases for the proposed technique to compare situational strengths and weaknesses to those of the parallel Lustre file system that is deployed on CLAIX at the RWTH Aachen. For this matter the I/O benchmark IOR and the metadata benchmark mdtest are used to simulate I/O patterns that typically are observed for I/O intensive applications. The results are presented and put into context to assert advantages and recommend the best use of such an on-demand NVMe storage in a HPC environment.

Contents

List of Figures

List of Tables

1 Introduction

With ever increasing scale of High Performance Computing (HPC) systems and steadily advancing CPU technology, the I/O subsystem is starting to become more of a concern for the performance of large HPC clusters [**?**]. About two decades have passed since the introduction of scalable distributed parallel file systems for HPC systems. They were designed to support high I/O throughput by allowing parallel access to multiple storage devices. Furthermore they are structured in a way that allows the owner to scale the size of the system according to the needs of its users by adding more storage devices. Parallel file systems have since been used to saturate the needs of HPC systems for scalability, flexibility and high performance I/O. The main component of parallel file systems are Hard Disk Drives (HDD), which offer high amounts of storage space and are affordable. Although HDD performance has improved over time, the rate is not comparable to those at which the computational power of CPUs has increased. While there are newer and more advanced storage technologies apart from HDD, they are still too expensive to supply the large demands for storage space of HPC users. Improvements for cheaper technologies are however not sufficient to match the development in concurrency and computational power.
Moreover the I/O workload on HPC systems is very bursty [**?**] with a huge number of concurrent jobs and deviating I/O intensity. Many of today's highly parallel and I/O intensive applications increase the pressure on the file system.

Modern storage device technologies like Non-Volatile Memory Express (NVMe) Solid State Drives (SSD) are able to handle many of the arising problems, but they are very expensive which is why it is not feasible to supply the needs of HPC users with this technology as large HPC I/O systems need many Petabytes of storage space. Furthermore, even when NVMe SSDs are directly used in centralized parallel file systems, the performance is also dependent on other components like the incorporated network. Additionally, the parallel file system has to be optimized for many types of I/O access patterns. For this matter I/O operations are typically separated in to two groups on parallel file systems, object data operations and metadata operations. Object data operations refer to requests like read or write, the common I/O operations to access or change the contents of a regular file. Metadata contains information about other files and metadata operations access and change this information. These two types of operations are handled separately by the file system and their performance has to be optimized individually. This is very difficult as parallel file systems have grown in complexity with increasing numbers of components and additional layers of system software, hindering administrators in localizing and

working on bottlenecks.
Various studies in the past have implied that on HPC I/O systems the performance of metadata operations is often neglected in favor of optimizations in throughput [**?**] [**?**] and cheaper storage space.

In summary, the two main bottlenecks of HPC I/O systems are: the comparably slow HDDs and the complexity of the parallel file system which introduces possible further bottlenecks and makes it difficult to optimize the system for all scenarios. Similarly, we find two reasons why deploying newer and more advanced storage devices is not a feasible way to counteract these bottlenecks: state-of-the-art technologies like NVMe SSDs are too expensive for the high storage space demands of HPC systems and even when they are used, the complexity of the file system remains a concern.
Therefore we feel there is a need to look for new strategies to ensure I/O will not become the most prominent bottleneck of large compute clusters.

This thesis seeks to investigate some of the weaknesses of parallel file systems and wants to put forward an affordable and feasible strategy to avoid these weaknesses while also helping to stabilize the file systems workload. Furthermore the aim of this thesis is to examine the proposed strategy to find advantages and scenarios in which it is useful.
In this strategy the main file system is supplemented with the state-of-the-art NVMe SSD technology as on-demand storage devices on compute nodes. The NVMe SSDs are setup with the parallel file system BeeGFS as completely separate 'secondary file system'. In this way they appear as one coherent storage and support HPC applications running on multiple nodes efficiently. This secondary file system is proposed to be used completely on-demand, offering a clean and idle I/O system for applications that request it.
Working as a clean and on-demand file system entails that the data that remains on it, after an application execution finishes, has to be addressed. While users want their simulation output, system adminstrators and users alike are also interested in keeping the data centralized on one file system to safely maintain it. Therefore the remaining data has to be transferred over to the main file system for permanent storage. However, this is beyond the scope of this thesis and has to be addressed in future work.
Instead the aim of this thesis is to find use-cases that are well suited to the advantages the proposed technique offers, as well as to find other use-cases that put extraordinary stress on the main file system and should be offloaded to the secondary file system. The goal is to exhaust advantages of the NVMe SSD technology, improving the performance of applications that hit the weaknesses of the main file system, and to possibly alleviate the main file system from stressful tasks.
To find use-cases where this supplement promises significant performance improvements and to determine which applications should make use of it, three common I/O access patterns are selected and simulated. An extensive series of tests is done

on a small scale to compare the proposed strategy to the running Lustre File System on CLAIX at RWTH Aachen.
For these tests the customizable synthetic I/O benchmark IOR, developed by LLNL and widely accepted in the HPC community, is employed to simulate the three selected I/O scenarios. To supplement the results the equally common metadata benchmark mdtest, developed by LANL, is executed with a multitude of settings to evaluate metadata performance. Finally the group of benchmarks that is proposed by the IO-500 [**?**] to compare different HPC I/O systems is run on a small scale.
The comparison of a running production system to a clean and idle small-scale secondary file system is a challenge as the main file system is constantly under pressure and its performance therefore is limited and inconsistent. To keep track of the deviation in performance, all tests incorporated multiple runs. Moreover, the limited and inconsistent performance are conditions that are regularly encountered in everyday-use of the main file system and with the way we propose to use the on-demand storage, these very different conditions for the two systems are exactly what is needed to simulate behavior in everyday-use.

At first related work is discussed in the upcoming Chapter 2. The following Chapter 3 introduces necessary background knowledge before Chapter 4 explains the approach that was taken for the tests. Subsequently, Chapter 5 presents and analyses the related test results. These results are then concluded and put into context in chapter 6 to evaluate performance and utility. Finally, recommendations and an outlook for the future is given in Chapter 7.

Contributions of the thesis This thesis has found use-cases that perform significantly better on the secondary file systems as well as use-cases that perform better on the main file system and assert the strengths of a large parallel file system. These findings could be used to derive a small set of I/O behaviors with the goal to distinguish whether an application is likely to experience advantages on the secondary file system. Furthermore this thesis was able to exhaust the performance limitations of the used NVMe SSDs and the gained information could be used to offer a prediction on the behavior of the secondary file system on a larger scale or with different hardware, like SATA SSDs.

2 Related Work

Hongzhan Shan and John Shalf utilized the IOR benchmark in their study of two HPC platforms in 2007. They compare the performance of the GPFS on the Power5/Federation cluster at the National Energy Research Scientific Computing Center (NERSC) and the Lustre file system on the Cray XT4 at Oak Ridge National Laboratory. For the comparison they discuss effects of caching and influences of multiple settings like transfer sizes, concurrency and I/O interfaces [**?**].
In a further study Hongzhan Shan et al. show how IOR can simulate application I/O behavior and therefore can close the gap to application benchmarks like MADbench2, VORPAL-I/O and FLASH3-I/O. Many of the essential parameters of IOR are explained and it is illustrated how the behavior of each of the three application benchmarks can be mimicked with IOR [**?**].
Richard Hedges et al. employed IOR, mdtest and simul to test the performance of the Lustre file system at Lawrence Livermore National Laboratory (LLNL) in 2005 while it was pushed towards production use in their compute center. As IOR is also developed at LLNL, this study in part was reason for them to add and improve on features of IOR. The benchmarks were used to constantly evaluate the performance of the Lustre file system while it was regularly updated [**?**].
Samuel Lang et al. conducted another evaluation of a parallel storage system on the Intrepid, a IBM Blue Gene/P system at the Argonne Leadership Computing Facility (ALCF). For this the IOR benchmark and the Metarates benchmark were used to measure scalability before application benchmarks, inlcuding MADbench2 and BTIO, were utilized to provide more realistic results for applications [**?**].
Carns et al. described and examined the utility and performance impact of the Darshan I/O profiler by running experiments on the two IBM Blue Gene/P systems at ALCF, Surveyor and Intrepid. They conclude that Darshan has a negligible performance impact on most applications and can provide information of paramount importance when tuning I/O performance [**?**].
In another paper Huong Luu et al. studied the I/O workload of the Intrepid, Mira and Edison systems of ALCF and NERSC by analyzing the logs of the then by default for all users of NERSC and ALCF deployed Darshan I/O profiler. I/O patterns and performance are discussed as well as platform wide runtime and workload concerns [**?**].
Sadaf R. Alam et al. discuss concerns with metadata on modern and future parallel file systems. Experiments are done with mdtest and Metarates on GPFS and Lustre with an experiment setup that includes SSD devices to improve metadata performance. It was concluded that technological hardware improvements may not address future metadata issues adequately as hardware limits by far could not be

reached, possibly due to software limitations. However, they still reported a improvement by a factor of 2 to 4 in targeting the SSD devices over their currently deployed technology [**?**].

3 Background

In this chapter, the necessary background knowledge is provided to understand the approach and results that will be discussed in later parts. The underlying technology of HPC I/O systems is introduced, its structure explained and the two file systems that are relevant for this thesis, Lustre and BeeGFS, will be discussed in more detail. Finally the Non-Volatile Memory Express technology that is utilized for the proposed secondary file system will be presented.

3.1 Parallel File Systems

We first take a look at the underlying technology of typical I/O subsystems on a HPC system.
Modern HPC systems use what is called a 'parallel file system' to handle the large amount of I/O requests occurring on any of their compute nodes. Parallel file systems are an extension to conventional distributed file systems and therefore often are referred to as distributed parallel file systems. A distributed file system allows storing data across multiple storage devices while still presenting it to the user as one coherent storage. This is achieved by a software protocol (eg. NFS) that handles I/O calls, making multiple storages appear as one by adding a software layer that controls and manipulates the access to any data across all devices to represent a coherent file system to the user [**?**]. Parallel file systems most importantly also allow parallel access to files, making it possible for multiple nodes to write to or read from one file at the same time. This functionality is paramount in a HPC environment where concurrency is one of the keys to performance. Most HPC applications run across multiple nodes with many of them running on tens or even hundreds of nodes. To provide high throughput in a case where multiple processes across different nodes write to a single file the parallel file system 'stripes' the data, partitioning it to store it on multiple storage devices. With this, it is possible to achieve very high throughput numbers, as one application theoretically is able to use the cumulative available bandwidth of all storage devices. Besides data-striping, there is an abundance of methods employed to ensure first and foremost performance but also coherency, integrity and security [**?**].

There are multiple examples of parallel file systems that are worth mentioning, here are just a few:

BeeGFS, developed at Fraunhofer Institute for Industrial Mathematics with a focus

on ease of use as well as high flexibility and scalability [**?**]. BeeGFS is free of charge and used on many top HPC systems.

IBM's GPFS, a result of research projects by IBM that today is marketed as 'Spectrum Scale' and also is used on top HPC systems including the Jülich Supercomputing Centre (JSC).

The Lustre File System, an open-source platform that is used on the RWTH Compute Cluster as well as on a majority of the Top500 systems.

The next sections will take a closer look at two of these distributed parallel file systems, the Lustre File System and BeeGFS and will also introduce the NVMe technology.

3.2 Lustre File System

After a short overview of distributed parallel file systems, this chapter describes the specifics of the Lustre File System in more detail as this is the file system that will be used for later tests in this thesis.
Lustre is an open-source platform focusing on performance and scalability with a client-server model, hence it is used on many big compute clusters. According to Lustre's official website, there are currently installations with over 50 petabytes storage space that also have reported throughput numbers of more than one terabyte per second [**?**]. In Figure 3.1, a simplified structure of the Lustre File System is illustrated. At the bottom we see the Lustre Clients, which in numbers can range from a single node to ten or even hundred thousand nodes. Each client is connected to a high performance network that, depending on the hardware used, can transfer up to 100 Gigabit per second on each connection. With a well structured topology this can yield huge throughput numbers. The data is transferred through the network to a number of servers that can be divided into three groups:

Management Server The Management Server handles the configuration of the Lustre File System and its components. It stores all necessary information and file system registries to the Management Target. 'Target' is the term used for the conventional storage devices behind any of the servers. Typically, only a single Management Server is needed.

Metadata Server As the name indicates, the Metadata Server is responsible for all metadata operations. It manages the file systems namespace, inodes and index. All relevant metadata is then stored to the Metadata Target. Depending on the scale and workload of metadata operations happening on the system, multiple Metadata Servers can be implemented to share the workload and therefore improve

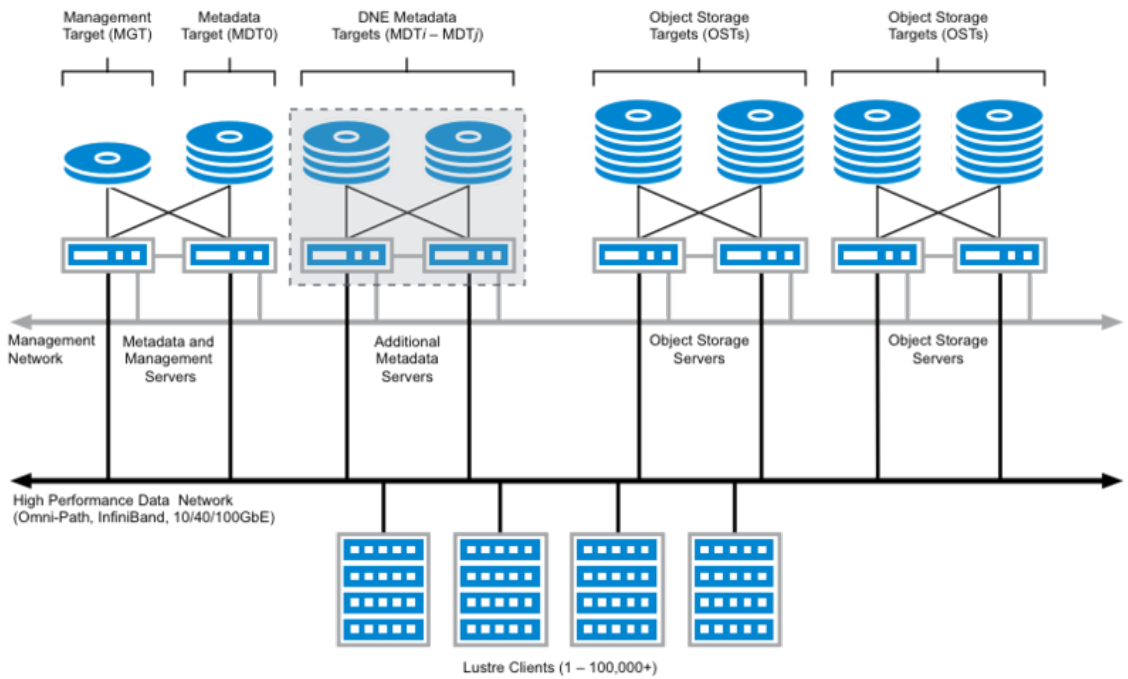


Figure 3.1: Structure of the Lustre File System [?]

concurrency and performance.

Object Storage Server Last but not least, the Object Storage Server is the main component that does the heavy lifting. Any actual data that users want to be written persistently goes through the Object Storage Server which then stores it on a Object Storage Target. To sufficiently handle the workload on big clusters many Object Storage Servers and Targets are used. CLAIX makes use of, in total, 32 Object Storage Targets.
Lustre tries to efficiently divide the workload among the servers and for parallel access to a single file uses the aforementioned data-striping: The Object Storage Servers partition the data and each partition is stored on a different Storage Target concurrently. In a theoretical case a single application running a large amount of processes across multiple nodes could access a single file and by data-striping Lustre is able to make use of the bandwidth of all Storage Targets at once, making it possible to almost fully saturate the cumulative bandwidth (some overhead of course is involved).

One of the main goals of Metadata Servers and Object Storage Servers is to separate different types of I/O tasks to be handled more efficiently: metadata operations typically are very small in size and intensive in terms of operations per second, while most normal data operations involve larger blocks of data and therefore are more

throughput intensive. By having metadata and normal data blocks handled on separate servers, each component can be optimized specifically for their designated task and work on it more efficiently.
To scale up the performance of such a file system, allowing more concurrent throughput, one can add more Object Storage Servers and Targets to the system or in some cases improve the network. Improving the metadata performance can be done similarly by increasing the number of Metadata Servers and Targets, however the performance is also heavily dependent on operation latency and on the possible number of simultaneous operations. The structure of a distributed parallel file system like Lustre can not offer latencies comparable to those of local storages as data additionally has to travel through a network, has to be handled by another server and then is transferred to the storage device. Furthermore the number of possible simultaneous metadata operations is closely connected to the metadata server executing them and the limitations of the storage device it works with.
Another paper [**?**] has also indicated that metadata performance is very sensitive to the systems overall workload, hence the performance can regularly drop significantly in an environment with a fluctuating workload. Further tests at the Swiss National Supercomputing Centre have also shown that their achievable metadata performance is 'orders of magnitude' apart from the limitations of the involved hardware (Solid State Drives). They suspect the software design of Lustre as a potential obstacle, while pointing to several papers [**?**][**?**] that implement 'intelligent software strategies' to improve metadata performance. These strategies however are not available for Lustre or other common parallel file Systems.

3.3 BeeGFS

BeeGFS, formerly Frauenhofer Gesellschaft File System, is a parallel file system for HPC systems developed at the Frauenhofer Institute for Industrial Mathematics and focuses on performance as well as ease of use and management. It was first used outside the Fraunhofer Institute in 2009, since has grown in acceptance and is used by many systems in the Top500 today. Metadata and object data are separated with a Metadata Service and Storage Servers. The general structure of BeeGFS is similar to that of Lustre but it also allows to incorporate local storage devices rather than completely concentrating on a domain of storage devices that is separated from compute node [**?**].

3.4 Non-Volatile Memory Express

The Non-Volatile Memory Express (NVMe) tackles current and potential future hardware limitations of I/O systems, including those in a HPC environment, with

high attainable throughput and I/O operations per second.
While the name NVMe already commonly is associated with a PCI Express (PCIe) based storage device, NVMe actually refers to the interface that is used to connect a Solid State Drive (SSD) via PCIe without any further manufacturer-specific drivers. The definition for NVMe by NVM Express, Inc. is the following: 'NVMe is an optimized, high-performance host controller interface designed to address the needs of Enterprise and Client systems that utilize PCI Express-based solid-state storage' [**?**]. It is driven by a consortium of many of the leading companies in the industry for storage solutions and was first introduced in March 2011 with its version '1.0'. In June 2017 they released the latest version, '1.3'.
The main goal of this interface is to fully utilize the capacities of the PCIe connection with PCIe based SSDs. With the NVMe interface taking big steps towards this goal and the PCIe connection improving constantly, the newest NVMe driven SSDs offer maximum bandwidths in the region of 2000 Megabyte/s for writing and around 2500 Megabyte/s for reading. Comparing that to today's conventional SATA SSDs, they offer almost 4 times higher bandwidths. Referring to random writes, the discrepancy is similar for I/O operations per second (IOPS), around 500.000 on NVMe SSDs versus 90.000 on SATA SSDs [**?**]. Although modern HDDs fall even further behind, it has to be pointed out that a HDD offers by far the cheapest storage space (in terms of cost per GB).
This also is one of the main reasons why SSDs and especially NVMe SSDs are barely used in large HPC I/O systems. Buying many Petabytes of storage space in the form of SSDs or NVMe SSDs is far too expensive with the current pricing.

4 Approach

After the most important background knowledge has been gathered, this chapter discusses the approach that was taken to face the challenge of evaluating use-cases for the secondary file system. Three common I/O access patterns are explained and concerns with caching are discussed as well as how it was handled. Finally a set of tools is introduced that is utilized to simulate the three patterns and conduct an extensive series of tests.

4.1 File I/O patterns in HPC

To conduct useful benchmarks with the goal of evaluating the NVMe SSD based secondary file system, each benchmark should model common I/O behavior of HPC applications. As we often encounter very common I/O behavior for many applications on HPC systems, we derive a set of three access patterns that tries to represent a broad range of application behavior.

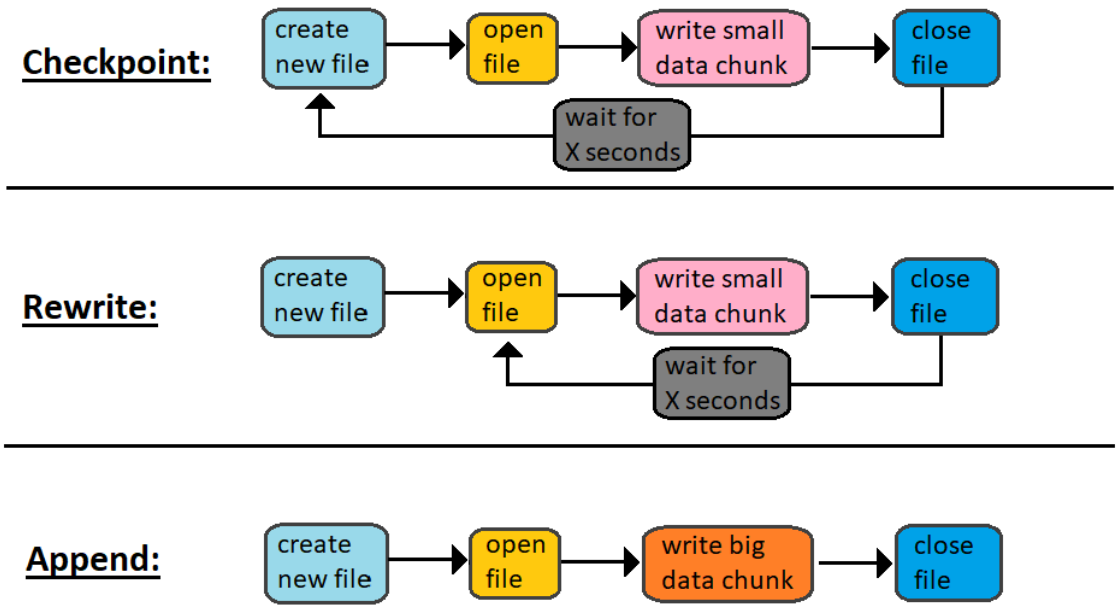


Figure 4.1: The three access patterns used in the thesis

Figure 4.1 illustrates the behavior of these three patterns. The first pattern is Checkpoint, an abstraction of the very common I/O behavior of scientific applications. It simulates an application running with multiple processes that repeatedly writes small files (eg. for potential restarts). Accordingly, this pattern uses an iterative behavior. In a file-per-process case each process opens a new file and writes a small chunk of data to it. The file is then closed and after a given delay each process starts the next iteration. Shared files are handled similarly but only one process creates a new file which then is opened and written to by all processes. Again the file is closed and after a certain amount of delay the next iteration starts.
The second pattern is named Rewrite and is an abbreviation of the Checkpoint pattern. It models an application that accesses and rewrites the same file(s) multiple times. In the file-per-process case each process creates a file at the start of the first iteration. Instead of creating an unique file in every following iteration, the content of that file is rewritten during each iteration. When a shared file is used, only one process creates a file at the start which then is used by all processes and during all iterations.
Finally, the third pattern is Append, which simulates an application writing a large file at the end of its execution. With file-per-process each process creates its own file and writes a large chunk of data to it. With shared file, a single process creates a file and all processes write a large chunk of data to this file. This pattern is not iterative, each process only writes one large consecutive block of data.
Although we simulate I/O behavior that in a similar way can be observed for many HPC applications with these three patterns, there are still large deviations in problem sizes, concurrency and I/O intensity amongst these applications that need to be addressed.
Previous studies on HPC I/O systems have illustrated that a big portion of jobs only issue I/O requests in the range of Kilobytes to a few Megabytes [**?**] and reach total data sizes of below one Gigabyte [**?**]. One of these studies also found that over 40% of its applications spend more time with metadata than with regular data [**?**]. Another related study pointed out that the vast majority of files found on two HPC I/O systems are below 64 Megabytes (94-99%) or even below 64 Kilobytes (43-58%), implying that many applications generate a big number of small files [**?**].
This information helped to determine fitting settings for each pattern. For Rewrite and Checkpoint, a total of six different file sizes per process was tested: 32 Bytes, 2 KiB, 64 KiB, 256 KiB, 1 MiB and 2 MiB. The delay between iterations was varied from 4 seconds to 30 seconds and 120 seconds. In case of the 4 seconds tests 32 iterations were executed, in other cases the test was limited to 16 iterations. Additionally a set of tests with the Checkpoint pattern was conducted, using either no delay, one second or two seconds of delay between iterations. These tests executed 128 iterations each and were limited to the smaller three file sizes of 32 Bytes, 2 KiB and 64 KiB.
The Append pattern utilizes a broader range and overall larger file sizes per process: 2 MiB, 4 MiB, 8 MiB, 16 MiB, 32 MiB, 64 MiB, 128 MiB, 256 MiB, 512 MiB, 1 GiB, 2 GiB, 4 GiB, 8 GiB.

The following section will give a short introduction to the concerns with caching and how it was handled in our tests.

4.2 Caching

To speed up I/O operations for the application calling them, the node uses its available memory space to buffer the data and gradually transfer it to the I/O system. Meanwhile, the application can continue instead of waiting for the data to be completely transferred to the file system. Therefore caching is an inherent problem when trying to reliably measure the performance of the I/O system.
Without any precautions against this, caching would mean that instead of measuring the real throughput and speed of the I/O system, we only examine the performance achieved by buffering the data in the memory.

There are three common ways to counteract caching that will be used for tests in this thesis:

File Sync The first and most commonly used precaution against caching is the file-sync operation. This I/O operation flushes all buffered data for a specific file to the file system, ensuring that the data is transferred to the file system immediately.

Memory Allocation Another way to counteract caching is limiting the memory that is available. Each node can only use free memory space to buffer data on it. When the free memory space is not large enough for the data that has to be stored, it cannot be buffered and instead has to be transferred to the file system directly. Therefore we can prohibit caching by allocating free memory space at the start of the test.

Read-back Finally, we can ensure that the data is on the file system by accessing it from a different node. When a process has finished its write operation and has buffered data in the memory, the data cannot be accessed by a process on a different node. This means that any process that tries to access (eg. read) the data that currently still resides in the memory forces the buffered data to be immediately transferred over to the file system to make the access possible. Accordingly, we use a process on a different node to read-back each the data that each process has just written.

Nevertheless, this thesis tries to simulate realistic behavior that in a similar way regularly is exhibited by applications. Therefore, in our initial tests we also measure the performance that an application experiences when caching is not prevented or limited in any way as there are many cases in which applications are able to make use of its benefits.

4.3 Benchmark Tools

This section now introduces the benchmark tools that were used and explains the way in which they were setup. The first subsection starts with the I/O benchmark IOR, explaining how the three patterns Checkpoint, Rewrite and Append are simulated with IOR and summarizing the variations of settings that are tested. Later subsections introduce the I/O profiler Darshan, the metadata benchmark mdtest and the IO-500 list as well as the associated benchmark suite.

4.3.1 IOR

IOR (Interleaved Or Random) is a synthetic I/O benchmark developed at the Lawrence Livermore National Laboratory and is intended to test the throughput of parallel file systems. It offers the use of different I/O interfaces and is highly customizable to simulate a multitude of behaviors [**?**].
It is widely used as a general benchmark for throughput and scalability of HPC I/O systems and due to its customizability even in studies that seek to mimick specific application behaviors [**?**] [**?**] [**?**] [**?**].
IOR supports the use of the standard POSIX I/O interface, but also high-level parallel I/O interfaces like MPI-IO, HDF5 and NETCDF. There are a large variety of settings to control the I/O behavior, including file and transfer sizes, file segmenting and file type (shared or unique per process). A subset of the possible options for IOR, limited to settings that were used in this thesis, can be found with a short description in Table 4.1. In this thesis IOR is used to simulate the three previously discussed access patterns with the mentioned variation of settings. For our tests with IOR, the used file structure is illustrated in Figure 4.2 for the shared file case. Within a segment each process writes a block (with size specified as blockSize) of data with transfers of a specified transferSize. The number of segments can also be controlled. This structure with multiple segments simulates a common way data is structured into datasets by HPC applications. In the file-per-process case the blocks of each process are simply stored in an unique file, so the file of every process only consists of one or multiple segments containing one block each.
All of the tests for this thesis were run with only one segment, instead only varying the blocksize to control the file size. Furthermore all tests use the MPI-IO API, collective I/O calls and concentrated only on a write test.
To simulate the three introduced patterns, only a handful of settings is needed: for the Checkpoint pattern, a number of repetitions (iterations) is run with the multiFile and interTestDelay option.
For Rewrite similar options were specified but instead of the multiFile option the keepFile and useExistingTestFile option was used.
Finally for Append none of those options were needed, the default IOR behavior with the options we specified for all tests was sufficient to mimic this pattern, so in this case only a different set of blockSizes and transferSizes was used to simulate the bigger data chunks.

Option	Parameter	Description
api	-a S	POSIX, MPIIO, HDF5, NCMPI
testfile	-o S	full name of the testfile
blockSize	-b N	size of continuous block written per task
transferSize	-t N	size of each transfer
segmentCount	-s N	number of blocks per task
numTasks	-N N	number of tasks participating
repetitions	-i N	number of times the test is repeated
writeFile	-w	conduct a write test
readFile	-r	conduct a read test
checkWrite	-W	check for correctness after write operation
interTestDelay	-d N	delay between repetitions
filePerProc	-F	use unique file-per-process
keepFile	-k	do not remove testfile after the test
useExistingTestFile	-E	do not remove testfile before write access
multiFile	-m	unique file per repetition
memoryPerNode	-M N	amount of memory to hog on each node
fsync	-e	perform fsync upon (POSIX*) write close
collective	-c	use collective I/O calls
reorderTasksConstant	-C	change task ordering by one node after each phase (eg. write)

Table 4.1: Options for IOR that were used in the thesis

Table 4.2 presents an overview of the options used for each of the three patterns. Both the file-per-process and the shared file case were tested as well as a range of five different process counts from 8 up to 96 processes. Processes were distributed evenly among nodes. For further tests, options like fsync, memoryPerNode, checkWrite and reorderTasksConstant are additionally deployed to eliminate caching behavior as explained in Section 4.2. The memoryPerNode option is used to allocate most of the memory on each node to limit the space available for caching. Additionally checkWrite and reorderTasksConstant are used to force each process to read back the file of a process on another node after the write operation is finished, ensuring that the file is on the file system.
Important to mention is that IOR currently is only setup to allow the fsync option with POSIX, therefore the associated code sections in IOR were slightly changed so that a file-sync is also implemented and usable for MPI-IO.
Initial tests showed that the output information of IOR is relatively scarce. Therefore, to get precise additional information, a common HPC I/O profiler was deployed for all IOR tests. This profiler will be introduced in the following section.

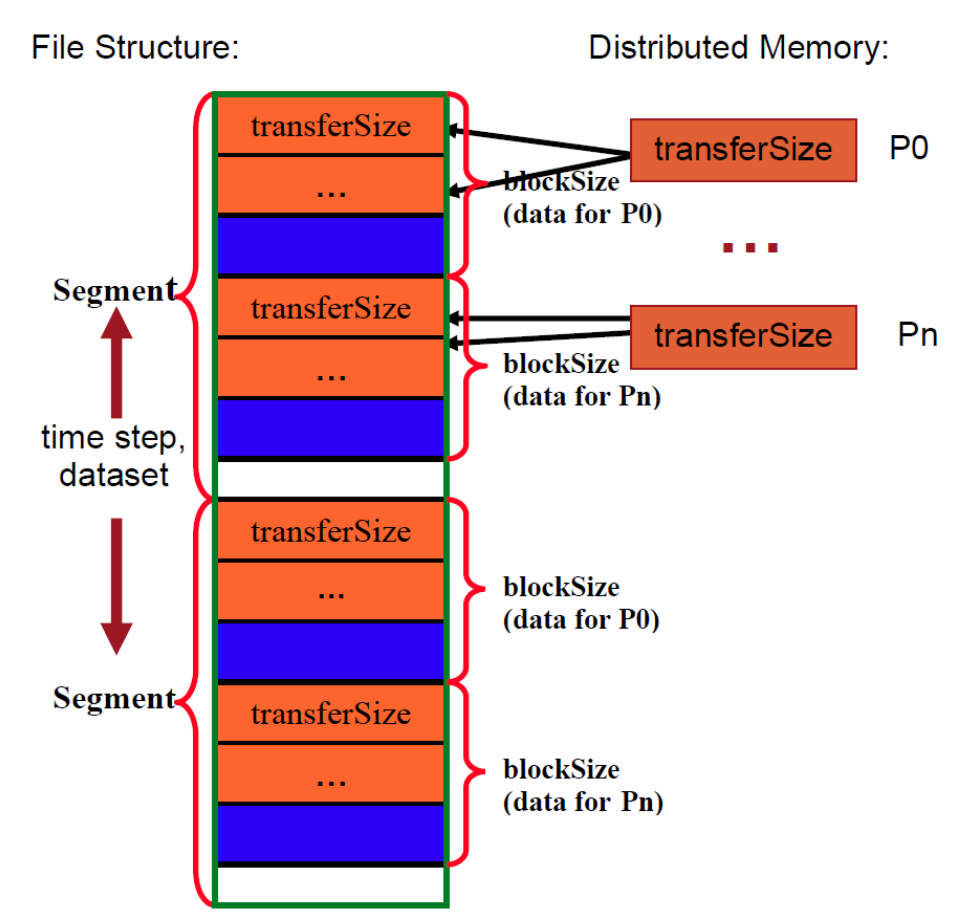


Figure 4.2: File Structure used by IOR in the shared file case [**?**]

	Append	**Checkpoint**	**Rewrite**
Shared Options	api=MPIIO, collective, segmentCount=1, writeFile		
Unique Options	—	multiFile	keepFile, useExistingFile
blockSize	2 MiB, 4 MiB, 8 MiB, 16 MiB, ..., 8 GiB	32 Bytes, 2 KiB, 64 KiB, 256 KiB, 1 MiB, 2 MiB	
repetitions	None	32, 16 (128) iterations	
interTestDelay	None	4s, 30s, 120s (additionally 0s,1s,2s)	
numTasks	8, 16, 24, 48, 96 Processes		
File type	shared file, file-per-process		

Table 4.2: Overview: Used IOR settings

4.3.2 Darshan

Developed at the Argonne National Laboratory, Darshan is a lightweight HPC I/O profiling tool that is centrally deployed on multiple compute clusters (eg. NERSC and ALCF). Darshan supports POSIX I/O as well as other APIs like MPI-IO, HDF5 and NETCDF. A previous study has tested '24/7' profiling on large compute clusters with Darshan, confirming that Darshan scales well and the overhead is minimal in most cases [**?**]. Another study has pointed out the upsides for application developers as Darshan often makes unexpected performance bottlenecks visible and also helps to find longstanding issues [**?**].
Darshan is provided as a library and can be preloaded at runtime, so there is no recompiling of applications needed. It creates a profile of the I/O operations of each process involved in the job and tracks cumulative operation counts and times for write, read and metadata on each file the process accesses.
For the tests with IOR in this thesis, Darshan was useful when trying to confirm the correct I/O behavior of the three set up access patterns. The most crucial use however was the exact timing of write, read and metadata operations for each process and file. As metadata performance is a big part of the concerns that were expressed in preceding chapters, it was important to get exact timings for each file that was accessed to get a picture of the behavior (especially stability).

4.3.3 Mdtest

To investigate metadata performance in even more detail, the common 'mdtest' benchmarking tool, developed at Los Alamos National Laboratory, is used. Mdtest is designed to run on parallel file systems and in an MPI environment. Each task creates, stats, reads and removes a specified number of files (and/or directories) and measures the cumulated operations completed per second [**?**]. Furthermore the size written to and read from each file can be set as well as a multitude of other settings can be controlled.
As the goal is a comparison of both systems in various situations, again a group of tests with varying settings was conducted to simulate different types and levels of stress on the system. After initial testruns a final group of tests was decided. Table 4.3 shows the used options and variations. Two file sizes were tested: 0 Byte files (empty file), which completely reside on the metadata server, and 3901 Bytes files, which force the files to be stored on regular storage targets. Another file size of 1024 Bytes was tested in initial tests, but results resembled those for 3901 Bytes, so it was left out in the final set of tests. As number of files 500 was tested as a minimum, with 1000, 2000 and 5000 to test higher levels of pressure. The number of tasks was again varied from 8 to 96. All tests worked through a total of 16 separate runs to track performance deviations. Furthermore all tests concentrated on the performance with files and used unique working directories per task. The performance while creating the respective directories (the directory 'tree') was measured, but no

Option	Used Setting
Bytes written/read per file	0 Bytes (empty file), 3901 Bytes
Number of files per task	500, 1000, 2000, 5000
Number of tasks	8, 16, 24, 48, 96
Iterations	16
Additional Options	Unique working directories, test only files

Table 4.3: Overview: Used mdtest settings

explicit tests on predominantly directories were conducted.
As in this case there are no simulated application behaviors, a look at what the IO-500 list proposes helped to decide on some options that are used.

4.3.4 IO500

To round out the test results, a downscaled version of the benchmark utilized by the IO-500 is run. Therefore this section introduces the IO-500 list, the benchmark they propose and how and why it was run on a smaller scale in this case.
The IO-500 list is pushed forward on the platform of the Virtual Institute for IO (VI4IO) by a team of researchers in the HPC I/O department. The VI4IO seeks to encourage research, collaboration and training on large scale storage systems [**?**].
The main goal of the IO-500 ranking is to refine and offer a group of benchmarks that can compare storage systems across any facility. Additionally, these benchmarks should monitor performance that is experienced by applications optimized for I/O as well as more randomly behaving applications and workloads with stress on metadata. With this goal, they propose a group of five mandatory benchmarks from which an aggregate score is generated to compare performances of different systems. These consist of two runs of IOR, 'IOR-hard' with completely prefixed settings that are supposed to be difficult to handle for the system and the second one, 'IOR-easy', with only a minimal amount of prefixed settings. The users are encouraged to exploit the systems limitations to the best of their abilities in this case.
Similarly, two runs of mdtest are conducted, 'MD-hard' preset to be hard for the system, 'MD-easy' to be tuned by the user to exhaust performance limits. The last test is a benchmark of the 'find' operation that also is to be set up and tuned by the user as they are encouraged to implement their own efficient parallel version of 'find' or use one of the included versions. An optional sixth benchmark is proposed with 'md-real-io' but it has to be set up by the user and results are not currently factored into the aggregate score.
This set of five benchmarks run in a particular order, starting with a 'Create' phase: IOR-easy and IOR-hard commence their write phase, followed by MD-easy and MD-hard create phases (all files are created and written to).
Next is the 'Access' phase: the 'find' operation is measured, IOR-easy reads its

associated files (note: ranks have been shifted beforehand), MD-easy does a 'stat', IOR-hard reads its files with shifted ranks and finally MD-hard does its 'stat'. MD-easy then removes its respective files and MD-hard performs a read on its files before removing them.
The prefixed MD-hard parameters are the source of inspiration for the settings that have been used for our own runs of mdtest, which have been explained in the previous section. All prefixed settings can be found in [**?**].
As a rule for submitting the results, to ensure reliability, the write and create phase of each IOR and mdtest run has to take atleast five minutes. As implied earlier, in the case of this thesis, the benchmark was downscaled, meaning this rule was not fulfilled. Instead a mark of roughly one minute create phases was set for the tests on the main file system and the same settings were then used on the NVMe SSDs. The reason for this is the used setup of NVMe SSDs and the permissions on the constantly in use Lustre file system, which did not allow for tests at any meaningful scale. However, as the primary goal was not to shoot for a (high) score that can be submitted to the IO-500, but to compare the performance of both systems on identical I/O behavior, the results are still worth a look. The IO-500 benchmark was run with 48 processes. Further settings that have been applied to achieve the one minute marks will be explained in accordance with the results.
The upcoming chapter on the results will start by discussing the mentioned small-scale hardware setup that was used as well as other important hardware details before presenting and explaining the results successively.

5 Results

This chapter first introduces the hardware setup on which the tests were conducted and presents all results successively, beginning with the findings for the three patterns simulated with IOR. Finally, the results for mdtest and our small scale IO-500 test will be discussed.

5.1 Hardware Setup

All benchmarks were conducted on the CLAIX system of the RWTH compute cluster, using four CLAIX-MPI nodes. The used NVMe SSDs are from Intels SSD DC P3600 series and are also plugged in on CLAIX-MPI nodes. Tables 5.1a and 5.1b show the technical specifications. The CLAIX-MPI Nodes utilize the Intel Broad-

Attribute	**Specification**
Number of Sockets	2
CPU Codename	Intel Broadwell EP
CPU Model	E5-2650v4
CPU Clock	2.2 GHz
Cores per Chip	12
Cores per Node	24
Memory per Node	128 GB

(a) CLAIX-MPI Nodes specifications [**?**]

Attribute	**Specification**
Capacity	2 TB
max. Seq. Write	1700 MB/s
max. Seq. Read	2600 MB/s
Latency	20 μs
Random Read	450.000 IOPS
Random Write	56.000 IOPS
Interface	PCIe NVMe 3.0 x4

(b) Intel® SSD DC P3600 specifications [**?**]

Table 5.1: Specifications of the used hardware

well architecture in form of the 12 core Intel Xeon E5-2650v4 on two sockets. Each node has 128 GB of DDR4 RAM.
The employed NVMe SSDs offer 2 TB capacity and 1700 MB/s of maximum write bandwidth. Important to note about this specific SSD model is the relatively low amount of possible random write operations per second (56000 IOPS) as the newer Intel Optane SSD 900P series already offers almost ten times as much (500000 IOPS). This likely is a big factor for the performance we can expect for metadata tests.
As discussed as use-case in Section 1, the NVMe SSDs are set up with a 'secondary' file system in BeeGFS and therefore are usable as one coherent storage when needed. All benchmarks are run on four nodes, equally distributing tasks among them.

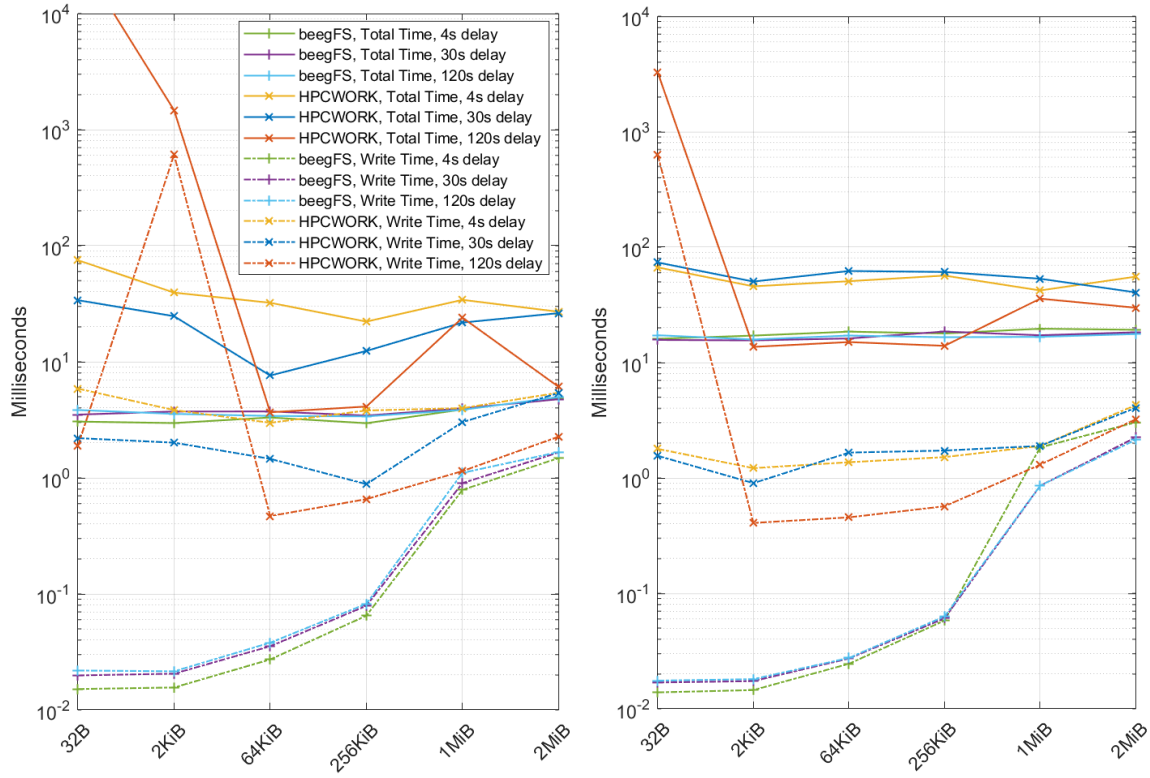
Therefore we can expect a maximum total write bandwidth of 6.8 GB/s (1.7 GB/s times 4) on the NVMe SSDs. The maximum bandwidth on the Lustre file system is hard to determine with multiple components at play. Each node is connected with a 100 GBit/s Omni-Path network and a total of 32 Object Storage Targets is present on the cluster. Furthermore there is 8 Object Storage Servers and one Metadata Server and one Metadata Target deployed. Tests by system administrators have confirmed achievable bandwidths on this system of close to 55 GB/s across a total of 64 nodes.
Finally, the upcoming sections will present the benchmark results, starting with IOR.

5.2 IOR

5.2.1 Checkpoint

The first IOR runs with the Checkpoint-pattern did not involve any specific precautions against caching. With the default settings described in Section 4.3.1 and file-per-process these initial runs yielded results that clearly indicated ongoing caching. In Figure 5.1 the total time taken and time taken for write operations is portrayed for 8 Processes (left) and 96 Processes (right). The x-axis shows the different file sizes with the y-axis indicating the time in milliseconds on a log-scale. As previously explained, these tests were run with 32 (for 4 seconds delay tests) and 16 (30 and 120 seconds delay) iterations. The time displayed is the average time across all iterations. A total of 12 lines are displayed with 'beegFS' referring to the tests on the NVMe SSD driven secondary file system and 'HPCWORK' referring to the main Lustre file system tests (in the following 'BeeGFS' or 'secondary file system' and 'Lustre' or 'HPCWORK' will be used to reference the two testsystems). Despite the ongoing caching, we can observe strongly varying performance on Lustre, especially for the smallest file sizes. On the other side the BeeGFS is very consistent in all cases. While the time taken hardly grows with the bigger file sizes on Lustre, the write time on BeeGFS grows by a factor of 100 from 32 Bytes to 2 MiB. Still, in both cases the total time is dominated by the time spent with metadata, which can be explained by caching on small files. Finally we do not experience any meaningful performance deviations for the different delays of 4, 30 or 120 seconds.

For the Checkpoint pattern Darshan provides by far the most information as each iteration uses a different file, which Darshan profiles seperately. In Figure 5.2 this information is used to monitor deviations more closely. All results for 32 Bytes (top) and 2 MiB (bottom) are shown with the different process counts on the x-axis. Times displayed are the median values and vertical bars display the 25/75 percentiles as well as the minimum and maximum time any process took for one iteration. The main plots show the total time and the small additional vertical bars that can be seen below indicate the median write time and respective 25/75 percentiles. We



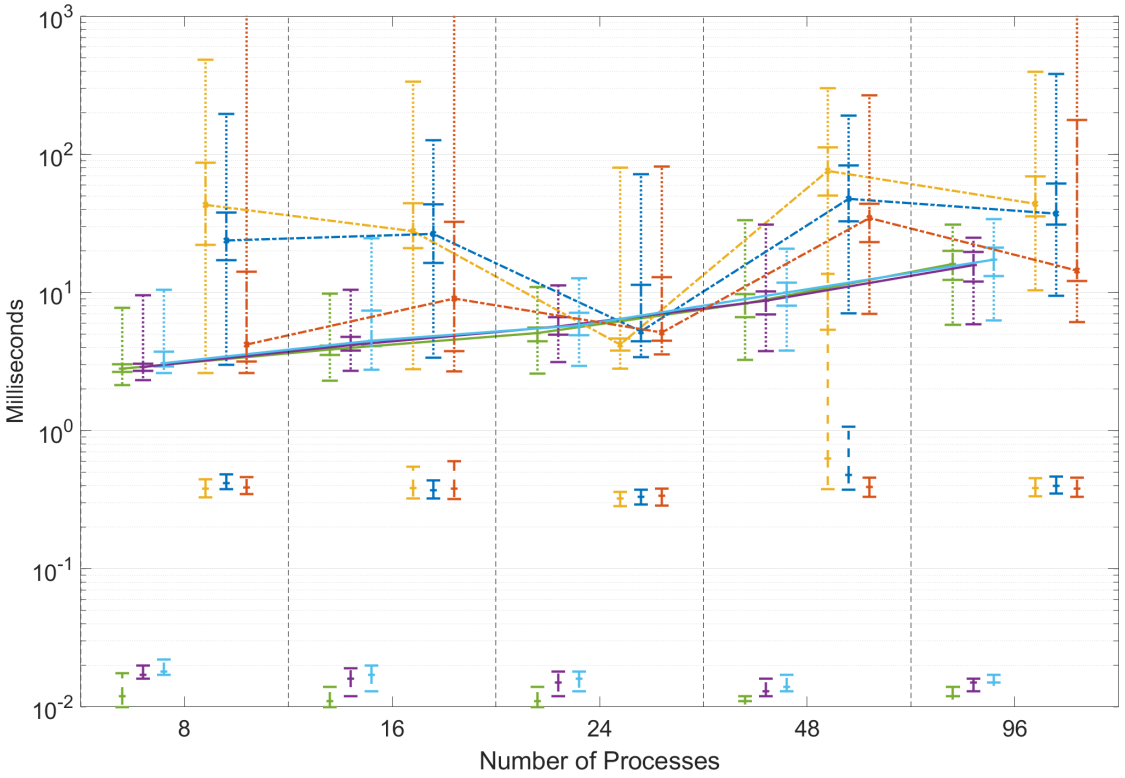
(a) 8 Processes

(b) 96 Processes

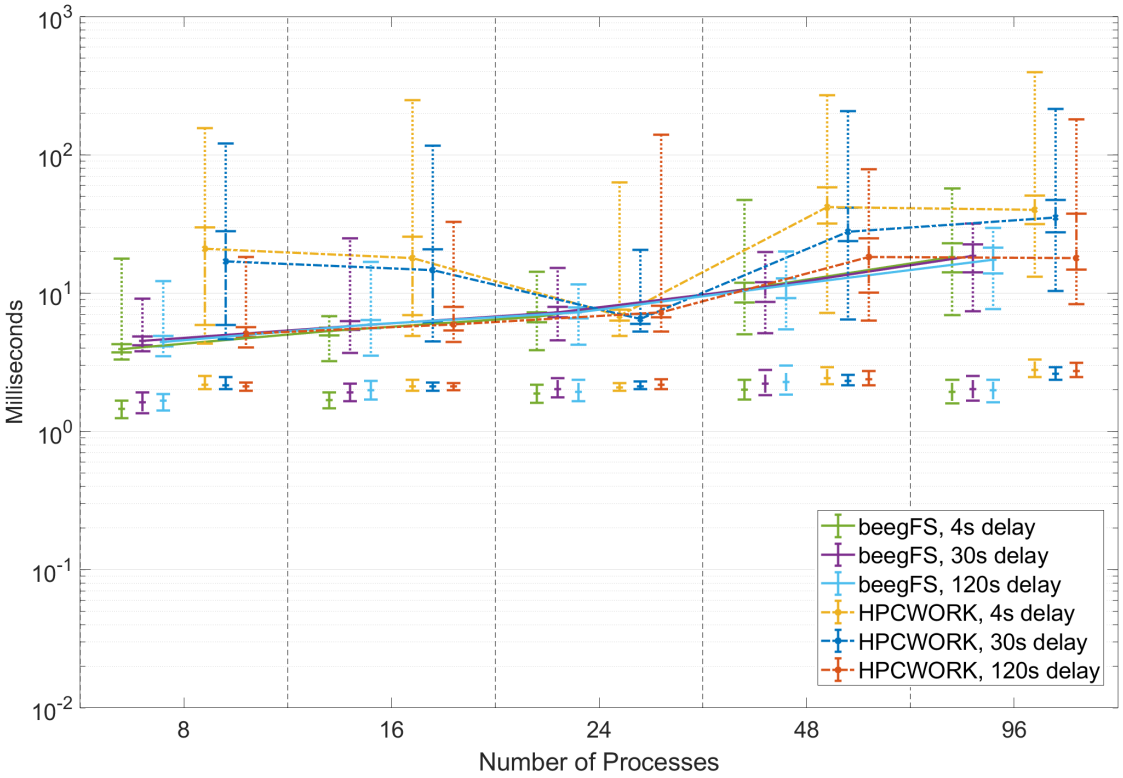
Figure 5.1: IOR Checkpoint-Pattern results for 8 and 96 Processes, with caching, file-per-process

again can make out by the minimum and maximum times of the total time that Lustre is very inconsistent for 32 bytes (this degrades steadily up to 2 MiB where it deviates less). With the associated bars showing the quite consistent write time we can also conclude that this instability is caused by metadata. BeeGFS overall deviates less but its performance degrades with the growing process count.

Eliminated Caching As a next step, caching was eliminated in another set of tests with the three precautions that were discussed before: using the 'fsync' option, using the 'checkWrite' option with preceding reordering of tasks ('reorderTasksConstant') to the next node (a process on a different node reads back the file) and using the 'memoryPerNode' option to allocate up to 98% of memory on each node to limit the space for caching in memory.
Figures 5.3 and 5.4 show the associated results. In these figures we now see HPC-WORK taking about one second for each iteration with the write time completely dominating the total time (Darshan includes the time for file-syncs in write time). For smaller file sizes we again see more deviation, but the median and average time is neither growing significantly with file size nor with the process count. BeeGFS



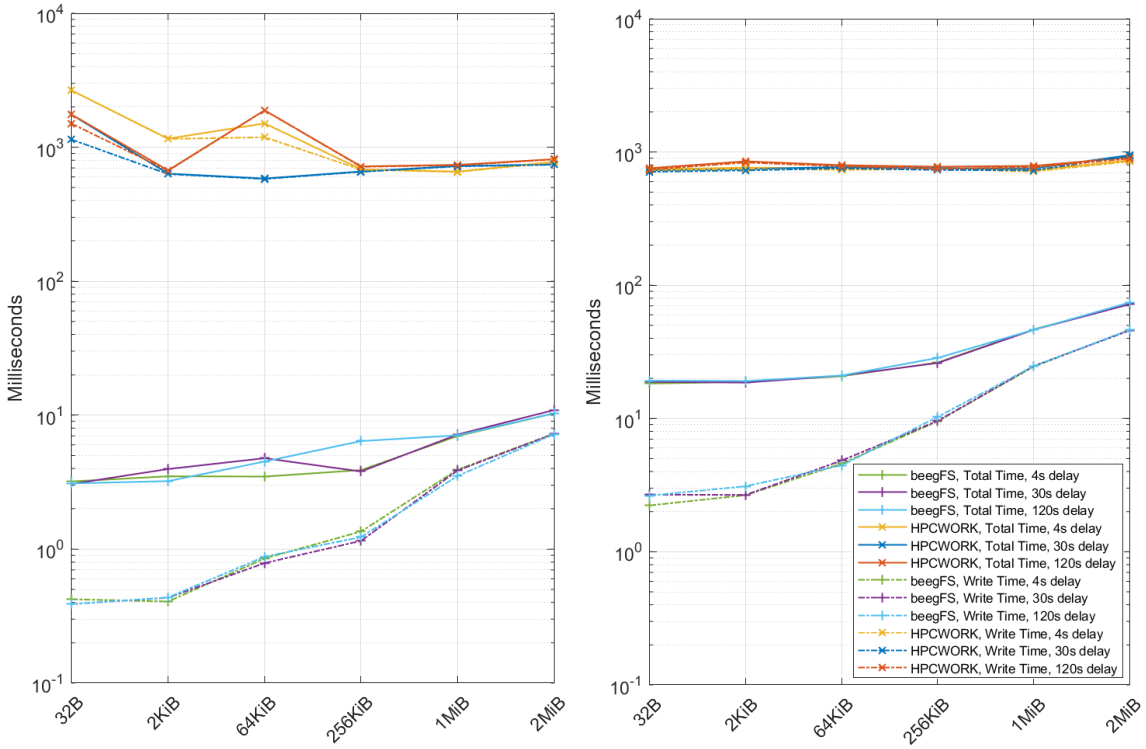
(a) 32 Bytes



(b) 2 MebiBytes

Figure 5.2: IOR Checkpoint-Pattern results for 32 Bytes and 2 MebiBytes, with caching, file-per-process

also is slower in this case, but the impact of caching is way more apparent on Lustre. For BeeGFS we experience growing times with both process count and file size, but even for 96 processes and 2 MiB file size BeeGFS remains faster by atleast a magnitude of 10. We again see the amount of delay between iterations have no meaningful impact. Figure 5.4 now also contains the read time (for all displayed results its the

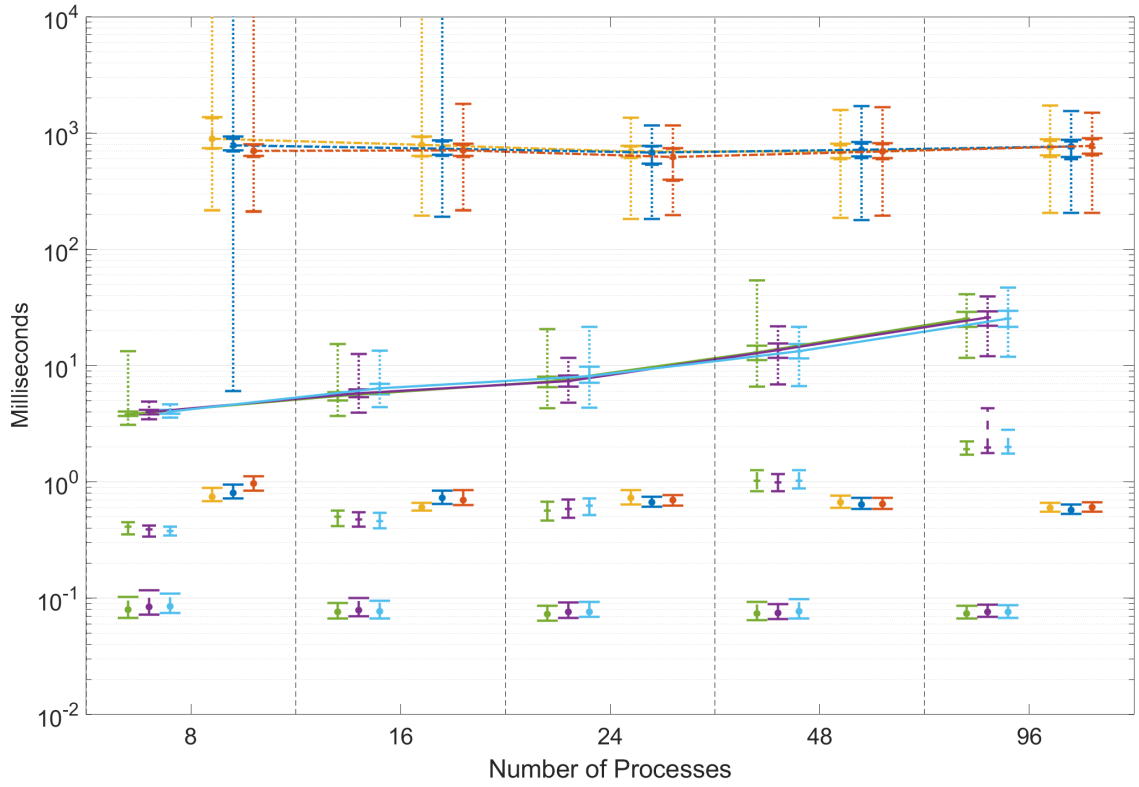


(a) 8 Processes (b) 96 Processes

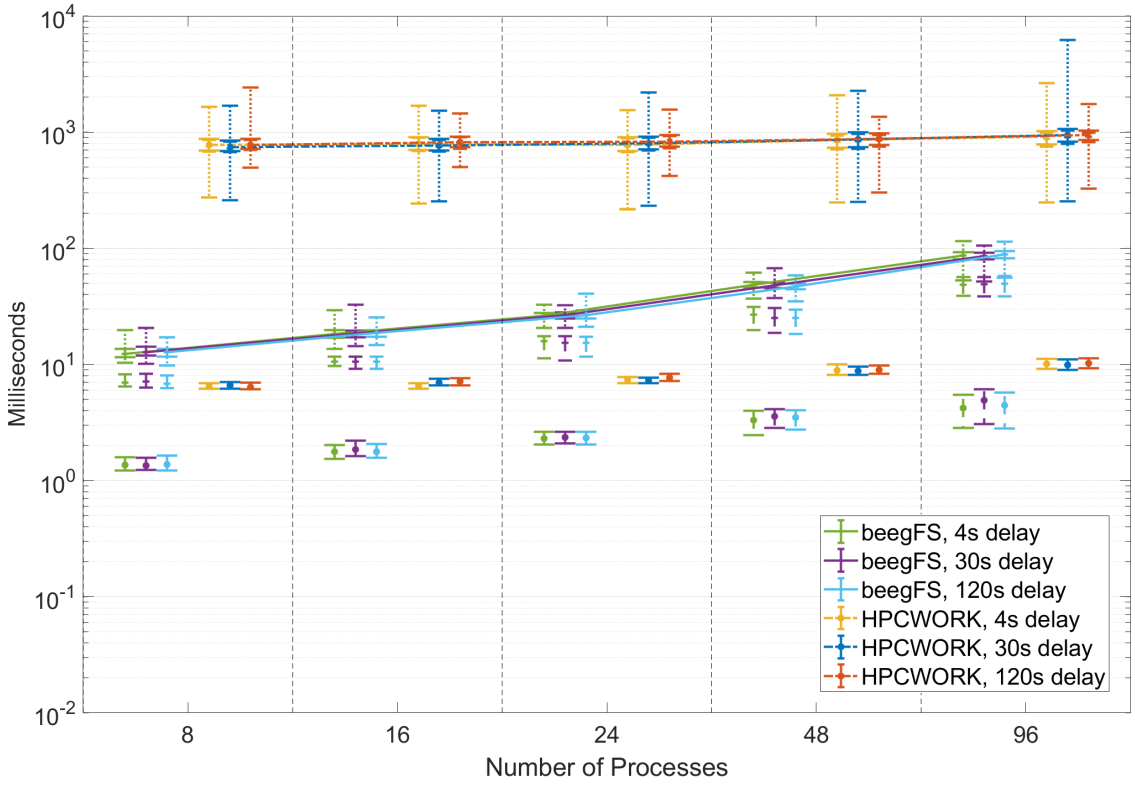
Figure 5.3: IOR Checkpoint-Pattern results for 8 and 96 Processes, caching eliminated, file-per-process

plot closest to the bottom) which refers to the time taken by the read operation of the checkWrite. This time is very small, therefore we assume the caching is sufficiently handled by our other two precautions. On Lustre the write-time dominates the overall time, hence the plots showing only the write-time are overlapping with the main plot and are not visible.

Shared Files The next set of tests is aimed at shared files. Again 'fsync' and 'memoryPerNode' is used, 'checkWrite' was skipped as it caused additional instability. As mentioned before, we expect caching to be handled sufficiently by the other two precautions. On the Lustre file system this group of tests, as well as a few others that will be covered in the upcoming chapters, encountered problems for 96 processes, getting stuck with a file-sync. The system adminstrators have been informed of the exact settings and situation this problem occurs in. For now, only



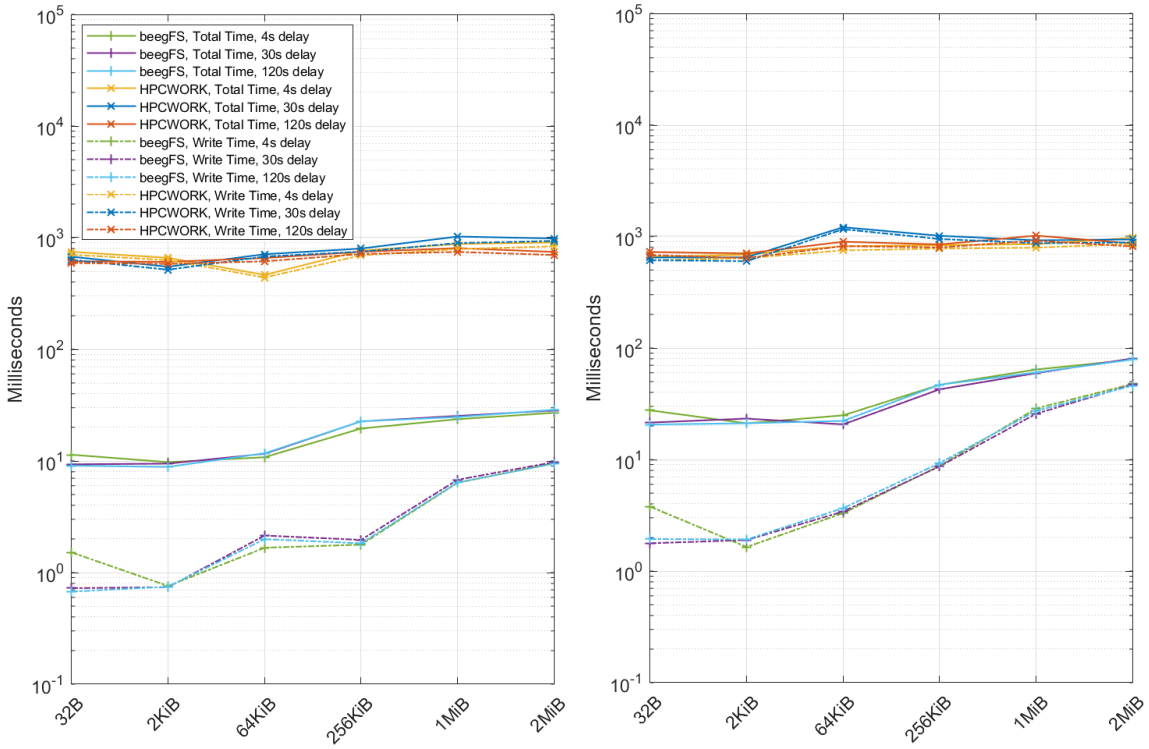
(a) 32 Bytes



(b) 2 MebiBytes

Figure 5.4: IOR Checkpoint-Pattern results for 32 Bytes and 2 MebiBytes, caching eliminated, file-per-process

results for up to 48 processes can be provided for Lustre. These results are portrayed

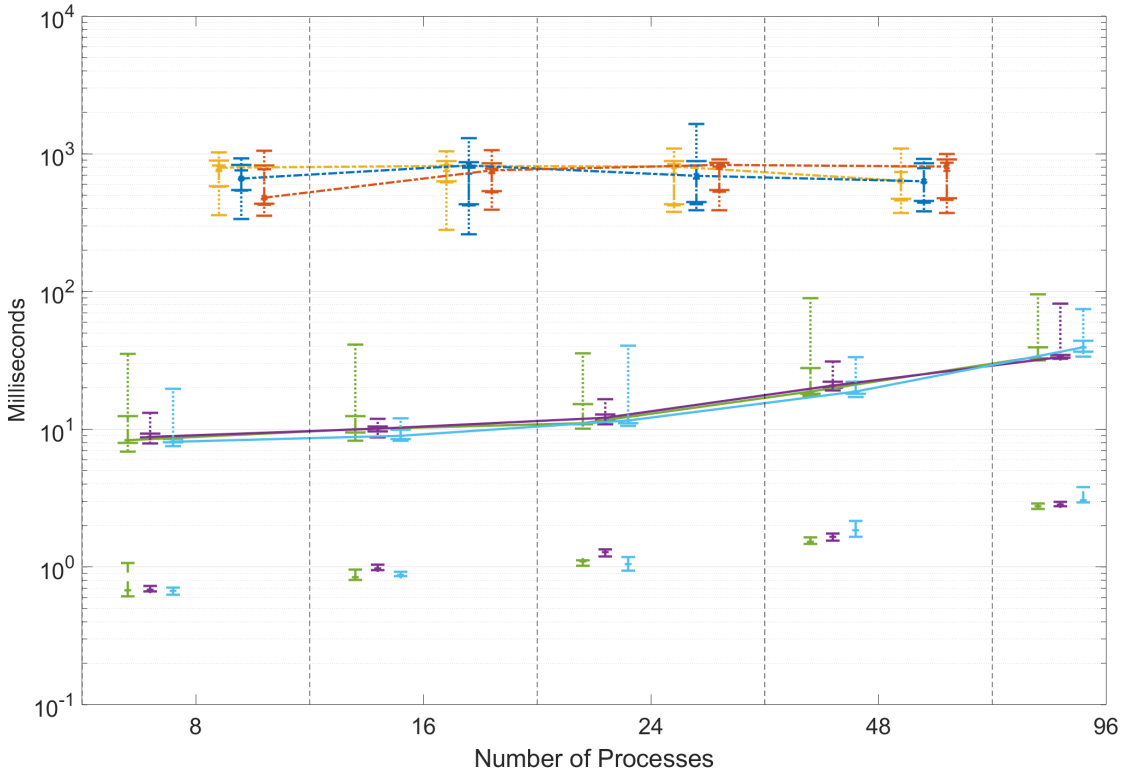


(a) 8 Processes (b) 48 Processes

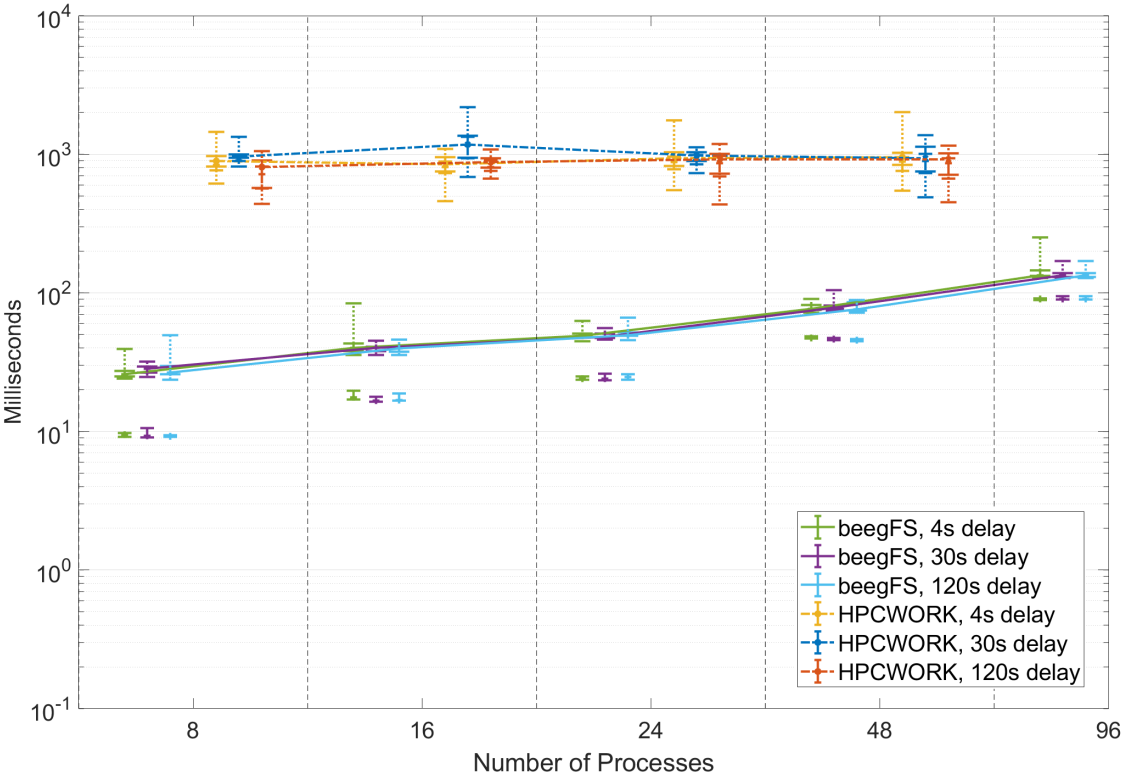
Figure 5.5: IOR Checkpoint-Pattern results for 8 and 48 Processes, caching eliminated, shared-file

in Figures 5.5 and 5.6 similar to before.
Figure 5.5 shows 48 processes on the right in this case. Even without results on 96 processes for Lustre there are clear trends visible. Overall shared files perform similar to file-per-process on Lustre, keeping its total time of about 1 second for all cases. BeeGFS is slower by about a factor of two compared to file-per-process but still far ahead of Lustre. The performance of Lustre again remains almost equal for all file sizes and process counts. On Lustre the write time also remains the dominating factor. For BeeGFS we see the same trends as before. The deviation for tests on Lustre as displayed in Figure 5.6 is significantly smaller with shared files. While for file-per-process Lustre experienced deviations (for minimum/maximum time) in the range of hundred milliseconds to multiple seconds, the deviation for shared-files is below a few hundred milliseconds in most test cases.

High Intensity Finally, another group of tests was set up to evaluate whether smaller amounts of delay between iterations (no delay, 1 second, 2 seconds) would have an impact on performance, as for 4 seconds and more there are no significant



(a) 32 Bytes



(b) 2 MebiBytes

Figure 5.6: IOR Checkpoint-Pattern results for 32 Bytes and 2 MebiBytes, caching eliminated, shared-file

differences. This set of tests again used the previous caching-precautions, file-per-process, 128 iterations and only the smaller three file sizes (32 Bytes, 2 KiB, 64 KiB). Figure 5.7 shows the results for 64 KiB which still indicate a negligible impact even

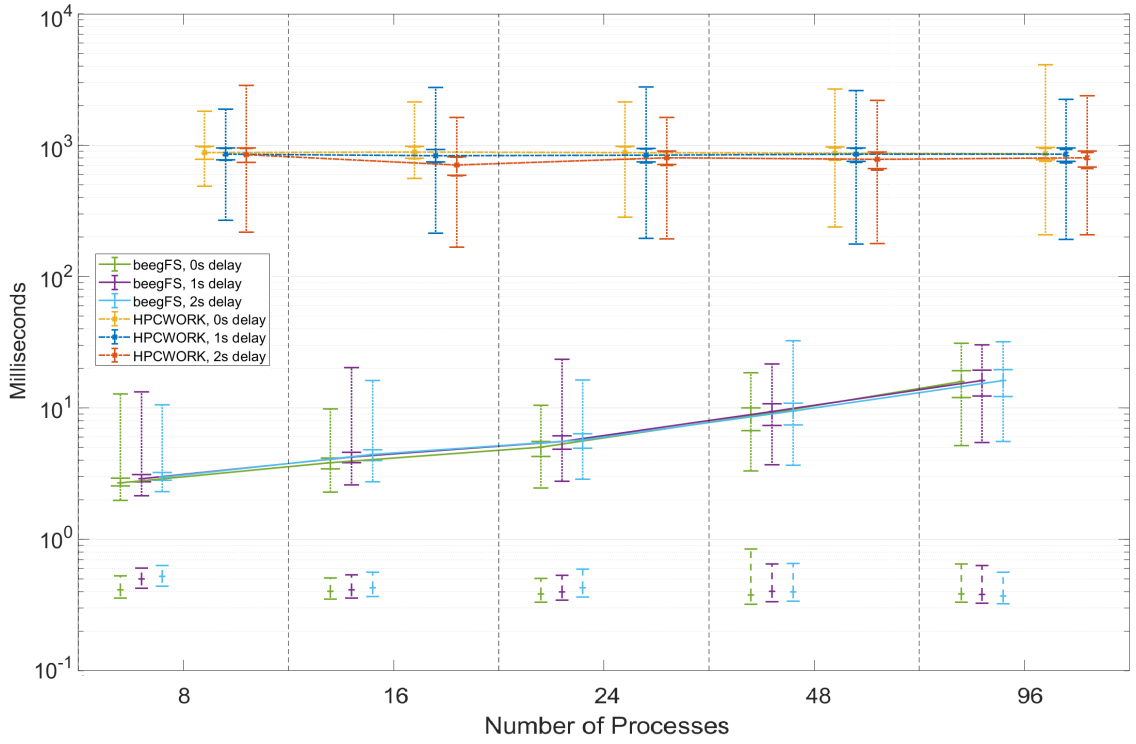
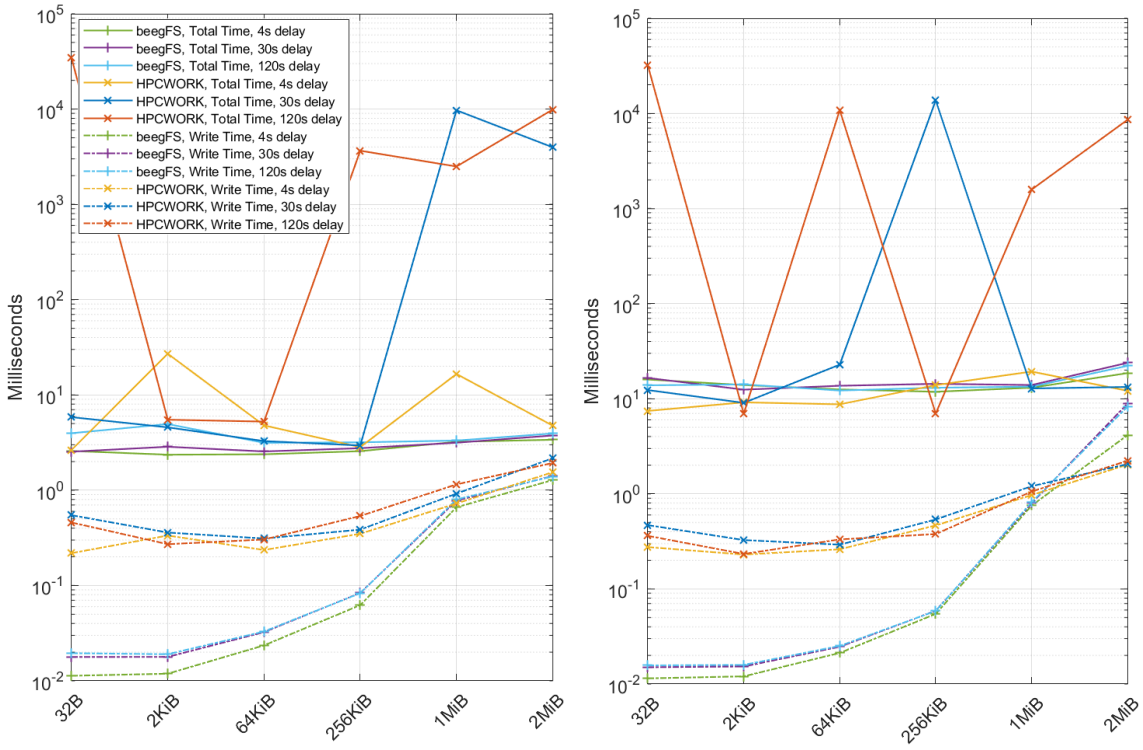


Figure 5.7: IOR Checkpoint-Pattern results for lower delays, caching eliminated, file-per-process

with no delay at all. Therefore there was no further examination in this direction.

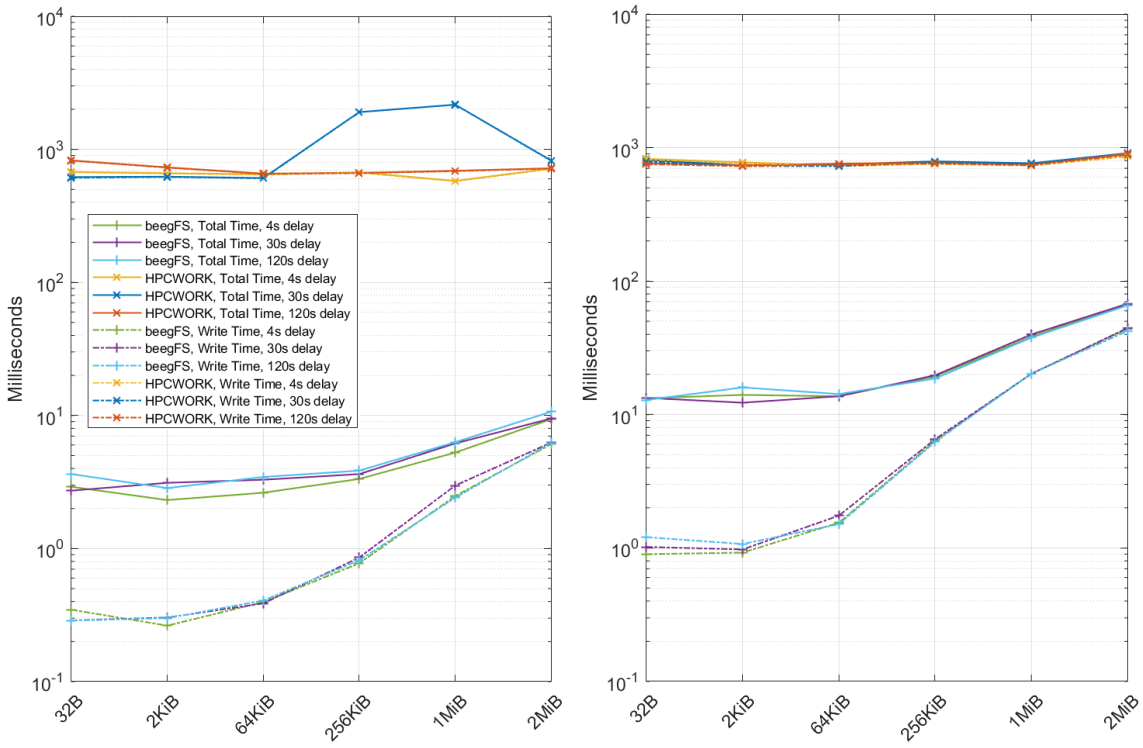
5.2.2 Rewrite

This section will now give a short look at results for the Rewrite pattern. The trend and overall performance of both systems in this case is very similar to that of the Checkpoint pattern. This is to be expected as the only difference is whether or not a new file is created and used each iteration. As for all besides the initial Checkpoint tests with caching the write time was by far the main factor on Lustre, taking out part of the metadata operations is unlikely to have a distinguishable impact.
Results for the first group of tests with caching are displayed in Figure 5.8. Besides an even more volatile performance on Lustre, the runs that performed well are barely faster than the respective ones of the Checkpoint pattern. For BeeGFS there also is no significant difference visible. Therefore the comparison of both systems in this case leads to the same conclusions as for the Checkpoint pattern: Lustre's metadata performance tends to be volatile while the secondary file system is quick and consistent. However, when Lustre behaves well and nothing prevents or limits



(a) 8 Processes (b) 96 Processes

Figure 5.8: IOR Rewrite-Pattern results for 8 and 96 Processes, with caching, file-per-process

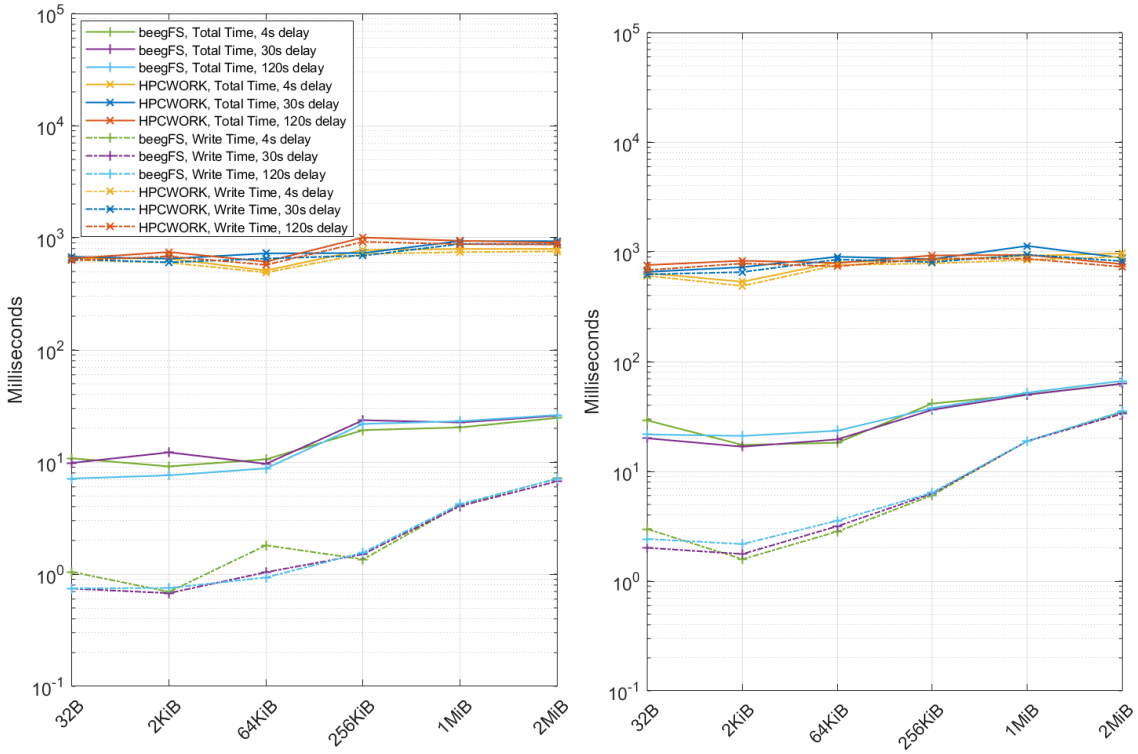


(a) 8 Processes (b) 96 Processes

Figure 5.9: IOR Rewrite-Pattern results for 8 and 96 Processes, caching eliminated, file-per-process

the use of caching, the performance experienced by the application can match up with BeeGFS.

Results for the tests with caching-precautions for both the file-per-process and shared-file case confirm this observation as their results perfectly line up with those illustrated for the Checkpoint pattern. Figures 5.9 and 5.10 contain the results for file-per-process and shared file (again missing 96 processes) with eliminated caching and underline the previous statement: BeeGFS performs better in all cases, HPCWORK takes about one second for each write operation and shared files perform very similar to the file-per-process case.
Hence the focus will be moved to the Append pattern which promises a completely different perspective on both systems as it puts more stress on throughput.



(a) 8 Processes

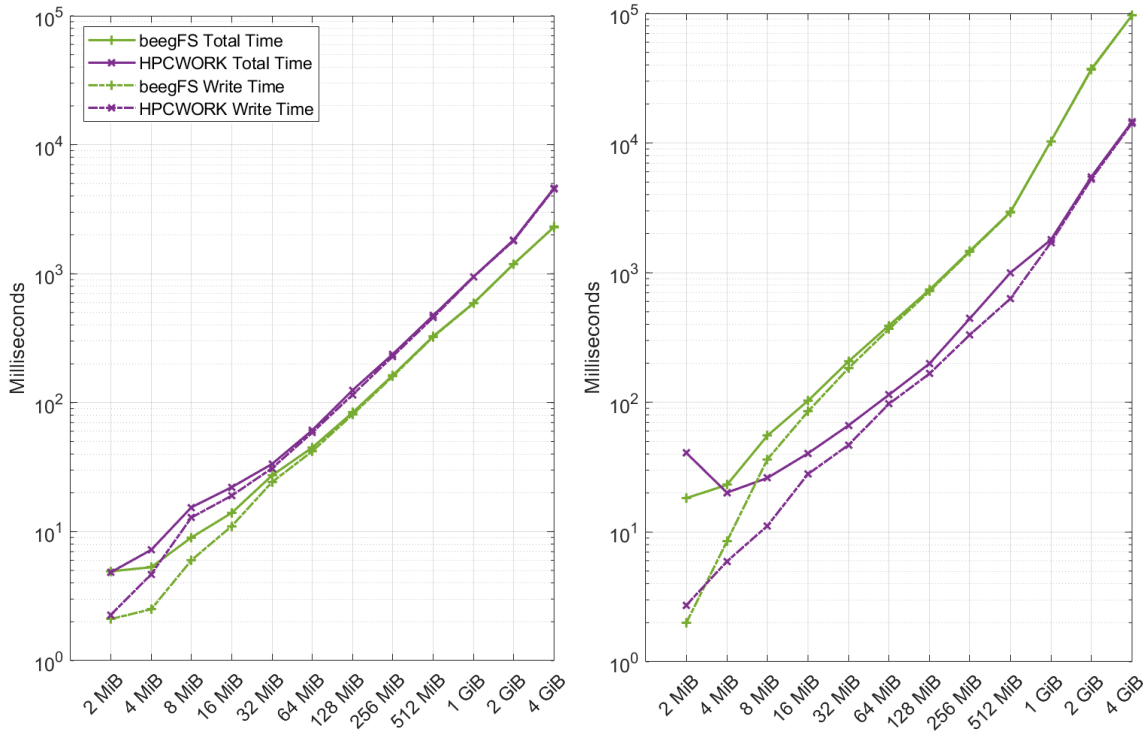
(b) 48 Processes

Figure 5.10: IOR Rewrite-Pattern results for 8 and 48 Processes, caching eliminated, shared file

5.2.3 Append

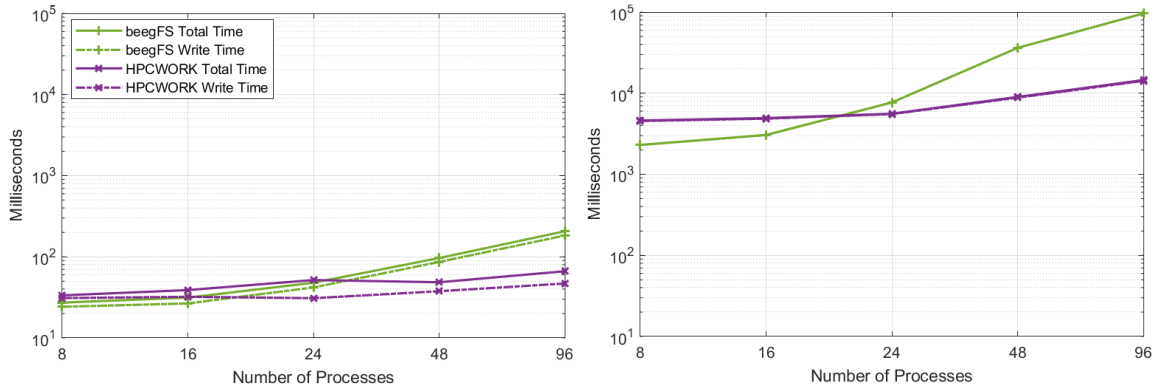
The Append pattern means less pressure in terms of metadata but more stress on throughput. With the previous results showing the biggest performance deviations due to metadata when caching was involved, a more consistent performance on Lustre would be expected with this pattern.

The first group of tests used twelve file sizes from 2 MiB to 4 GiB, no options against caching and file-per-process. Associated results can be found in Figures 5.11 and 5.12. In Figure 5.11 we see Lustre and BeeGFS performance being very close



(a) 8 Processes (b) 96 Processes

Figure 5.11: IOR Append-Pattern results for 8 and 96 Processes, with caching, file-per-process



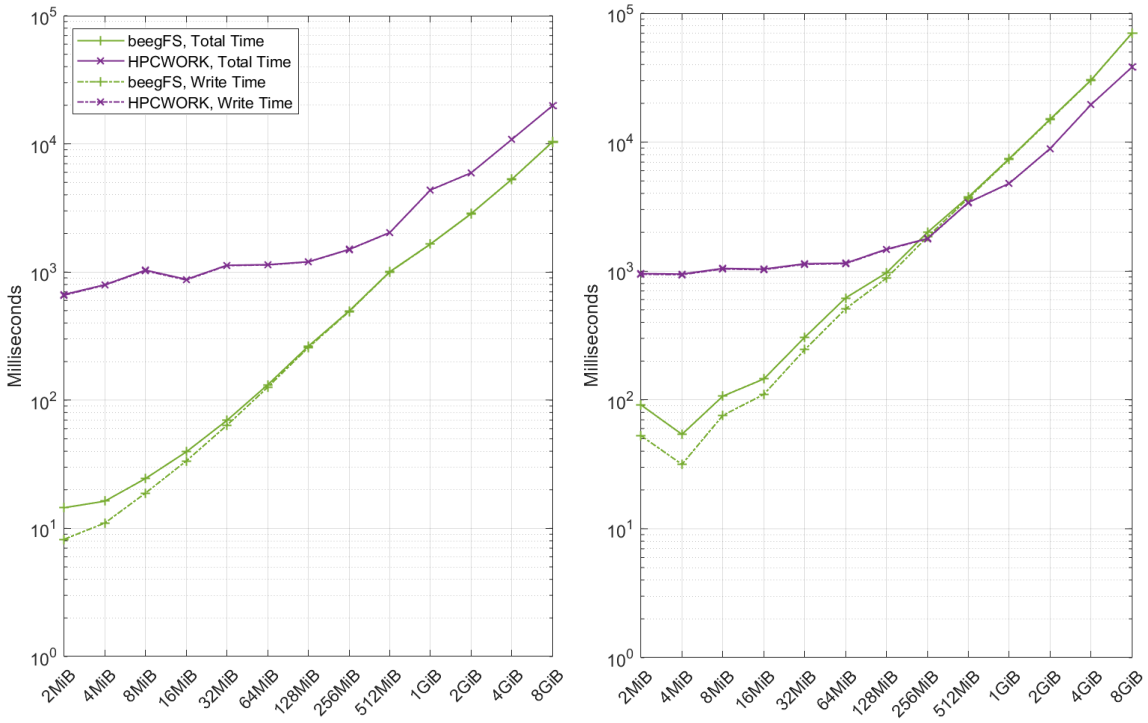
(a) 32 MiB file size (b) 4 GiB file size

Figure 5.12: IOR Append-Pattern results for 32 MiB and 4 GiB, with caching, file-per-process

when 8 processes are used, with a small advantage for BeeGFS. With 96 processes running, Lustre performs far better for file sizes above a few Mebibytes. For the biggest file size of 4 GiB Lustre is faster by almost a factor of ten. As to be expected, the write time closes in on the total time with growing file size. Figure 5.12 shows how the performance on one of the small file sizes and on the biggest file size change with the number of processes. In both cases, but especially for 4 GiB, the performance degrades more on BeeGFS when more processes are used. One factor that favors this behavior is the maximum total bandwidth of 6.8 GB/s that can be achieved across the four NVMe SSDs and most likely is reached for file sizes as large as multiple GiB and more than 8 processes. Therefore raising the process count can not yield any more throughput.
Another takeaway from these results is the much less volatile behavior with Append by Lustre apart from one outlier in Figure 5.11 for 96 processes and 2 MiB. As the write time is very small in this case, this outlier confirms the findings for the Checkpoint and Rewrite pattern, which indicated metadata performance to be inconsistent on Lustre, especially for smaller file sizes.

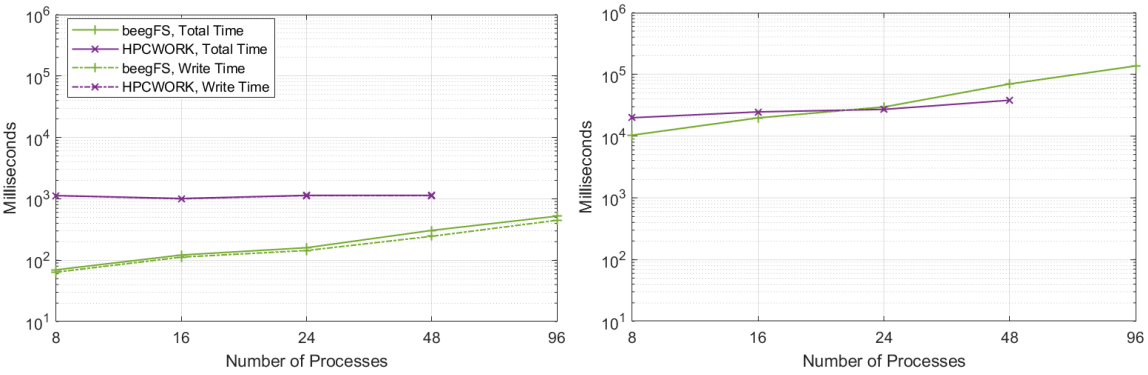
Eliminated Caching Similar to before, the next step is a testset with precautions against caching. For this group of tests another file size of 8 GiB was added. The 'fsync' and 'memoryPerNode' option were used as for previous tests. The respective results are illustrated in Figure 5.13 and 5.14. This was another case in which Lustre experienced problems with 96 processes, therefore the associated results are missing. This set of results paints a drastically different picture with caching eliminated. On small file sizes BeeGFS is faster by a factor that ranges from 10 to around 50 as Lustre takes about 1 second to write the small files, comparable to before in the Checkpoint and Rewrite tests with eliminated caching. This 'minimal write time' of Lustre was experienced for 32 Byte file sizes and remains until about 64 MiB, when the time finally starts to grow with the file size. With 8 processes BeeGFS outperforms Lustre even for the largest file sizes by a factor of two. However, with growing numbers of processes the performance of BeeGFS degrades faster and we see Lustre overtaking the BeeGFS at around 256 MiB. In Figure 5.14 we can see the performance of BeeGFS being better for less than 24 processes on 8 GiB, while both systems are about equal at 24 processes and Lustre is faster with more processes.

Shared Files Next, shared files are investigated for this access pattern. Tests are run with the same settings as before and results are shown in Figures 5.15 and 5.16. Once more Lustre encountered problems for 96 processes and associated results are missing.
The overall performance illustrated in these figures is worse in comparison to file-per-process, which on BeeGFS was also found for the Checkpoint and Rewrite patterns. However, the behavior for bigger file sizes (or in this case block sizes) is very inter-



(a) 8 Processes (b) 48 Processes

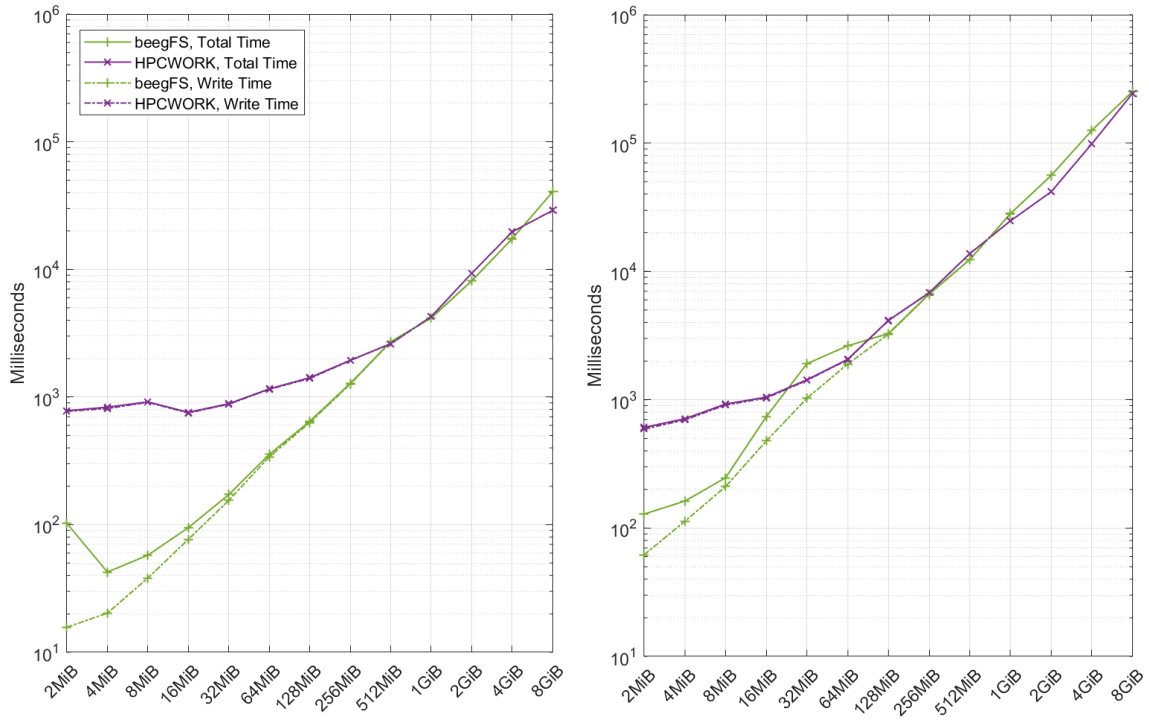
Figure 5.13: IOR Append-Pattern results for 8 and 48 Processes, caching eliminated, file-per-process



(a) 32 MiB file size (b) 8 GiB file size

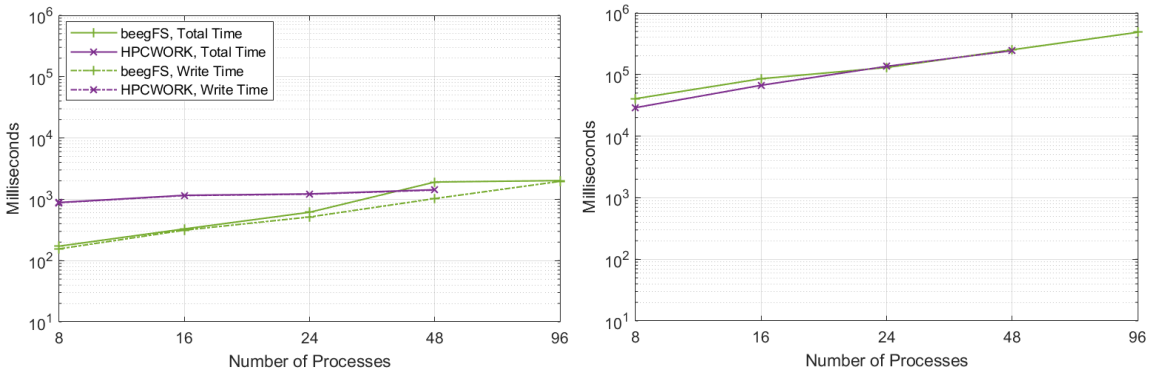
Figure 5.14: IOR Append-Pattern results for 32 MiB and 8 GiB, caching eliminated, file-per-process

esting. As visible in Figure 5.15, BeeGFS clearly asserts its advantages on smaller files but the performance of both systems gets closer with growing block sizes until we see almost identical times on both systems. This trend develops earlier for higher number of processes as BeeGFS drops in performance and is not far apart



(a) 8 Processes (b) 48 Processes

Figure 5.15: IOR Append-Pattern results for 8 and 48 Processes, caching eliminated, shared file



(a) 32 MiB block size (b) 8 GiB file size

Figure 5.16: IOR Append-Pattern results for 32 MiB and 8 GiB, caching eliminated, shared file

from Lustre even for small sizes. The graph for 32 MiB block sizes in Figure 5.16 illustrates this.

For the tests in which caching is counteracted, estimates for the achieved total throughput can be compared. These numbers are averages taken from the IOR out-

	Lustre		BeeGFS	
Block size	**8 Proc.**	**48 Proc.**	**8 Proc.**	**48 Proc.**
2 MiB	18 MB/s	75 MB/s	0.9 GB/s	1.3 GB/s
16 MiB	0.1 GB/s	0.7 GB/s	3 GB/s	4.7 GB/s
128 MiB	0.8 GB/s	3.9 GB/s	4 GB/s	5.3 GB/s
256 MiB	1.2 GB/s	6.4 GB/s	4.3 GB/s	5.8 GB/s
512 MiB	1.7 GB/s	8 GB/s	4.2 GB/s	6.3 GB/s
1 GiB	1.6 GB/s	9 GB/s	4.8 GB/s	6.5 GB/s
8 GiB	3 GB/s	9.4 GB/s	6.5 GB/s	6.4 GB/s

(a) File-Per-Process

	Lustre		BeeGFS	
Block size	**8 Proc.**	**48 Proc.**	**8 Proc.**	**48 Proc.**
2 MiB	20 MB/s	0.2 GB/s	0.4 GB/s	0.9 GB/s
16 MiB	0.2 GB/s	0.7 GB/s	1.5 GB/s	1.5 GB/s
128 MiB	0.7 GB/s	1.5 GB/s	1.7 GB/s	2 GB/s
256 MiB	1.1 GB/s	1.8 GB/s	1.7 GB/s	2 GB/s
512 MiB	1.6 GB/s	1.9 GB/s	1.7 GB/s	2.1 GB/s
1 GiB	2 GB/s	2 GB/s	2 GB/s	1.8 GB/s
8 GiB	2.3 GB/s	1.7 GB/s	1.7 GB/s	1.7 GB/s

(b) Shared File

Table 5.2: IOR Append-Pattern: Comparison of estimates for achieved throughput

put but should only be taken as a rough guideline as the performance is volatile. When the throughput is calculated with the time taken as logged by Darshan, the resulting values are found to be almost equal. The values can be found in Table 5.2. As the workload was distributed evenly among four nodes, the throughput per node would be one fourth of what is shown in the table.
The calculated values underline the advantages of BeeGFS on small files and when running only few processes. We see BeeGFS almost reaching its theoretical limit of 6.8 GB/s already at around 512 MiB and 48 processes which explains the better scaling on Lustre which still constantly increases in throughput when the number of processes and/or file size is raised. For shared files the throughput values reveal that the BeeGFS is peaking at about 1.7 GB/s in most cases, which would be the maximum write speed of a single NVMe SSD. Either it is a coincidence and shared files are just significantly slower and cannot reach higher throughput numbers even for bigger block sizes, or it could be that some tuning regarding striping of shared files to multiple storage devices is needed. For HPCWORK the maximum throughput is in a similar range and similar questions arise. Further examination might be needed but it would go beyond the scope of this thesis. Nevertheless concerns are expressed.
However the focus now will be switched to metadata performance with the results

of mdtest benchmark.

5.3 Mdtest

Section 4.3.3 explained the variety of settings that was tested with the mdtest benchmark. Mdtest measures the operations per second for each of six types of operations: file creation, file stat, file read, file removal, tree creation and tree removal. To conserve the results and make them presentable, figures were set up to show two types of operations each. In Figure 5.17 the operations per second (OPS or IOPS) for file creation and removal are displayed for the setting of 500 files per process. The y-axis shows the IOPS with the x-axis indicating the different process counts. A total of eight lines are shown as both file sizes, 0 Bytes and 3901 Bytes, are included. Vertical bars indicate the deviation in performance by marking the 25/75 percentiles. The green and light blue lines refer to the 0 Byte (empty file) tests and show a better performance on Lustre with IOPS in the range of 20000 while BeeGFS is at around 12000. For both systems the performance stays similar with growing process numbers. The tests with 3901 Bytes behave almost exactly the same on BeeGFS but present a very different picture on Lustre. File creation at first reaches similar values for process counts up to 24 but drops for 48 and 96 processes to below 5000 IOPS. File removal shows an opposing trend, being slow for low process numbers but improving until it reaches values comparable to those of the 0 Byte test at 24 and more processes. Figure 5.18 shows the IOPS of file stat and file read for the same test setting. We see BeeGFS being very fast with file stat, but lacking behind on file read. One more takeaway in the comparison is again the very volatile performance of Lustre, especially for the 3901 Byte case. For tree creation and tree removal, values are illustrated in Figure 5.19. In this case the secondary file system is outperforming HPCWORK on tree removal. Tree creation behaves similar on both systems for the 3901 Byte test but performed subpar with 0 Bytes on Lustre. Another thing to note is the drop in performance for tree removal on BeeGFS when reaching 96 processes.

When the number of files per process is raised, BeeGFS maintains the same performance for all operations besides tree removal, which drops in performance even for lower counts of processes. On the other hand Lustre struggles with multiple operations as for example file creation (3901 Bytes) drops to around 2000 IOPS even for 8, 16 and 24 processes. IOPS for file creation (0 Bytes), file removal (both sizes), file read (3901 Bytes) and tree removal (both sizes) also start to degrade heavily with growing process numbers when more files are accessed per process.
The subsequent performance for the maximum of 5000 files per process can be found in Figures 5.20, 5.21 and 5.22.
We see Lustre still outperforming BeeGFS on some operations for the 0 Byte tests but falling behind, in part heavily, for the 3901 Byte tests. Most notably file creation performs a lot worse on Lustre with 3901 Bytes. This drop in performance remained throughout all 3901 Byte tests whenever a total of around 20000 files or

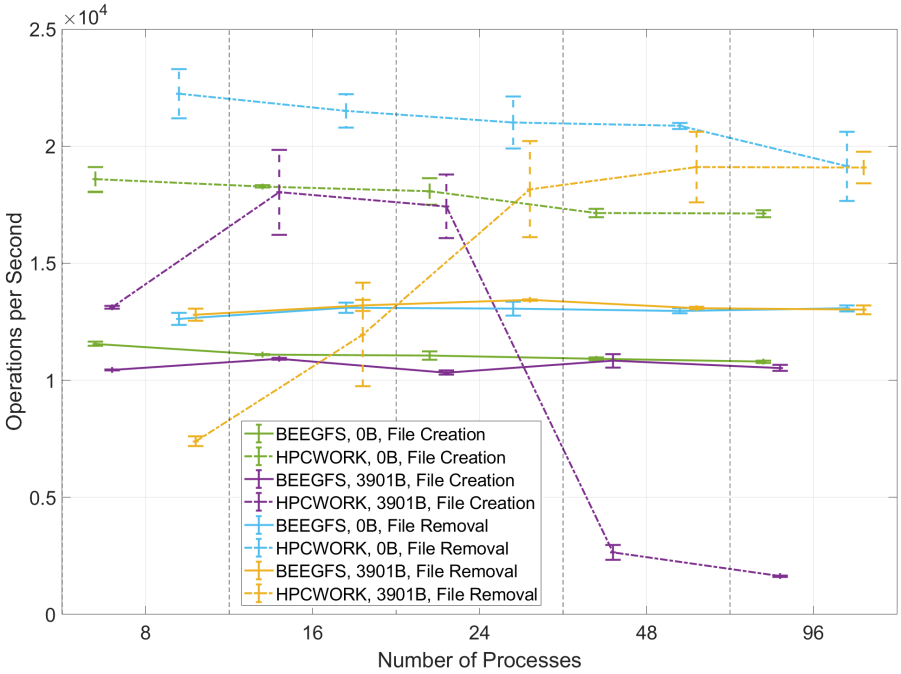


Figure 5.17: mdtest results: IOPS for file creation and file removal, 500 files per process

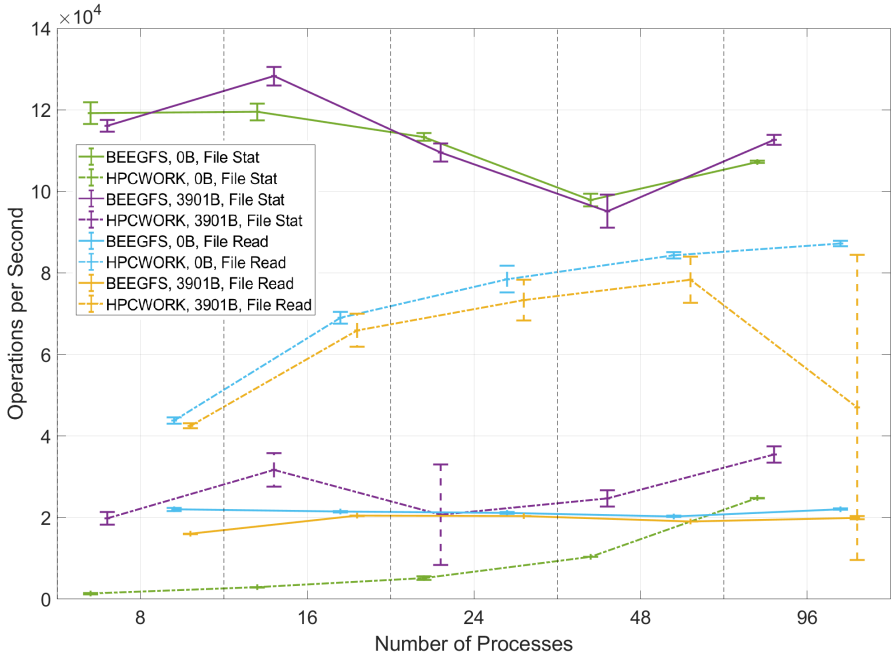


Figure 5.18: mdtest results: IOPS for file stat and file read, 500 files per process

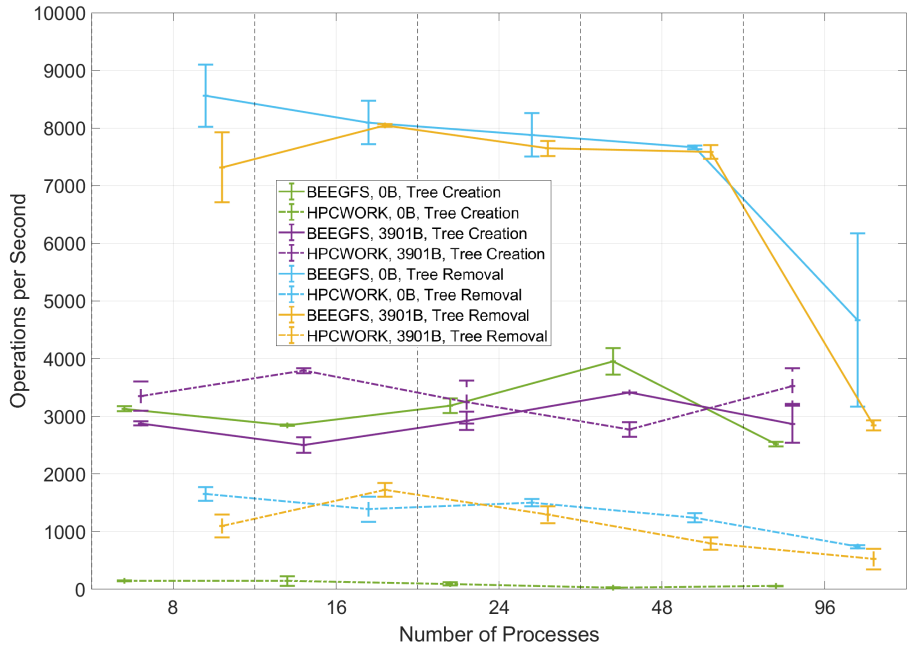


Figure 5.19: mdtest results: IOPS for tree creation and tree removal, 500 files per process

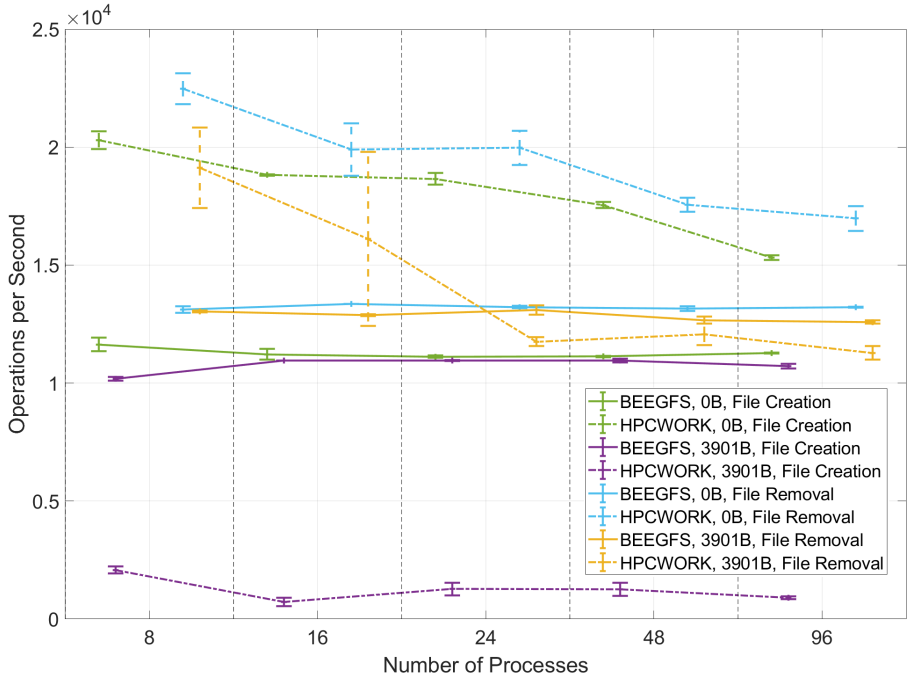


Figure 5.20: mdtest results: IOPS for file creation and file removal, 5000 files per process

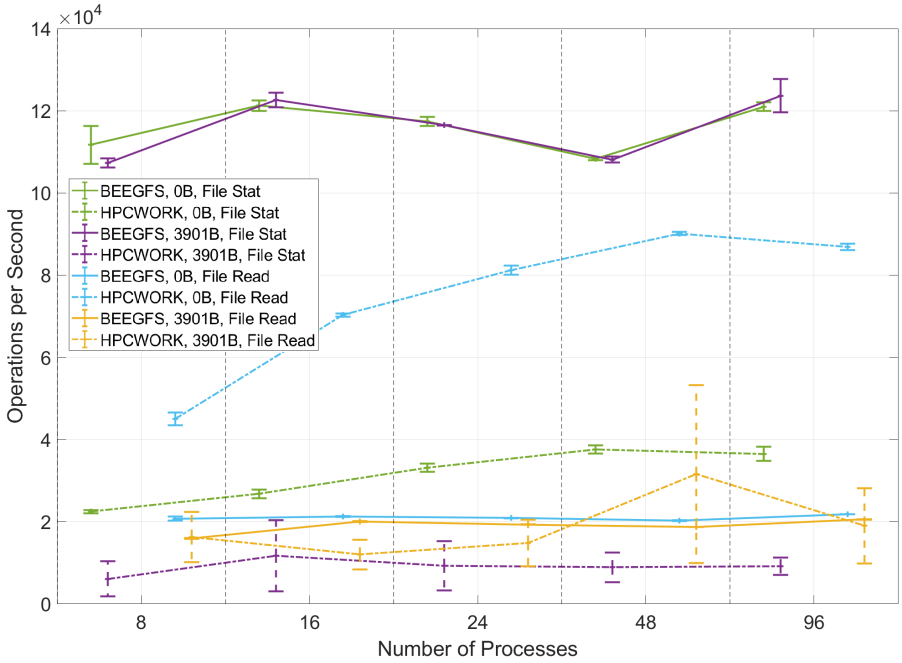


Figure 5.21: mdtest results: IOPS for file stat and file read, 5000 files per process

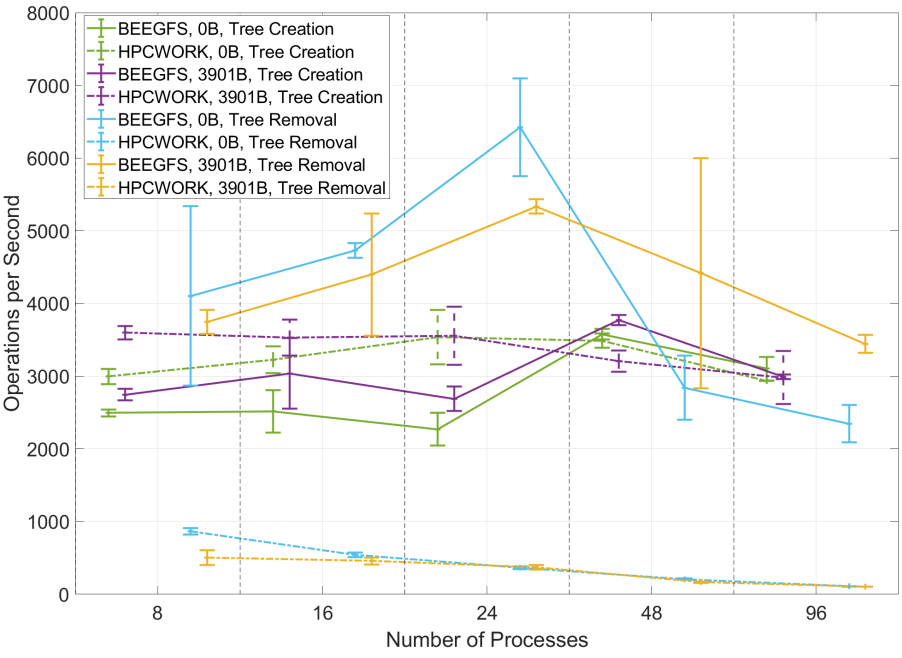


Figure 5.22: mdtest results: IOPS for tree creation and tree removal, 5000 files per process

more were involved (eg. 24 Processes, 1000 files per process or 48 Processes, 500 fpp). For 16000 files (eg. 8 processes, 5000 fpp or 16 processes, 1000 fpp) Lustre still achieved over 10k IOPS. It should be kept in mind as file creation is one of the most important metadata operations for real applications.
Overall Lustre delivered a surprisingly good metadata performance in regards to operations per second when looking at it independently. However the preceding IOR tests raised concerns whether this performance holds for real applications. The metadata performance of the NVMe SSD driven file system is consistent and is able to outperform Lustre when the workload surpasses certain levels. Nevertheless the performance is far from what the theoretical limits suggest, which was also found by a previous study [**?**]. In this context it also has to be pointed out that the employed SSD has a theoretical maximum of only 56000 IOPS for random write access, which is only about one tenth of what the newest NVMe SSDs promise [**?**] [**?**].

To finish off the results chapter, the findings of the small scale run of the IO-500 benchmark, which seeks to provide a broader comparison of all important I/O components, will be analyzed.

5.4 IO500

Chapter 4.3.4 explained the way in which the IO-500 benchmark employs multiple separate micro-benchmarks to come up with an aggregate score to compare different systems. The results that are provided through this benchmark include the write and read bandwidth for the IOR-hard and IOR-easy runs as well as the IOPS for create, stat and delete of MD-hard and MD-easy. Additionally the read IOPS were measured in MD-hard and the IOPS for the find-benchmark are provided.
It was mentioned before that the write and create phase of IOR and mdtest were set up to take about one minute. The settings that could be manipulated were applied as follows: IOR-easy wrote 16 GiB files as file-per-process; IOR-hard executed 1000 writes per process; MD-easy used unique directories, files only at leaves, empty files, 2500 files per process; MD-hard runs on 1000 files per process.
The test was run multiple times on both systems to ensure a reliable result. Table 5.3 shows the results of the runs with the median total score. In this benchmark BeeGFS with a total score of 10.79 clearly outperforms Lustre with a total score of 2.85. Furthermore the individual results of the micro-benchmarks confirm some of the earlier findings. On one hand Lustre handles easy, throughput-intensive (large files) access pattern very well, yielding higher throughput numbers where the NVMe SSDs reach their theoretical limits (the 7.7 GB/s on BeeGFS for IOR-easy also indicate possible caching as 6.8 GB/s should be the theoretical maximum). On the other hand Lustre struggles immensely with harder access patterns, where smaller transfers and file-syncs are involved. The BeeGFS in this case is still able to provide an acceptable performance.
The mdtest results for the create phase also confirm the problem on Lustre that was

	HPCWORK	**BeeGFS**
IOR-easy write	13.1 GB/s	7.7 GB/s
MD-easy create	2.3 KIOPS	30.3 KIOPS
IOR-hard write	0.03 GB/s	1.0 GB/s
MD-hard create	0.8 KIOPS	8.0 KIOPS
find	51.0 KIOPS	36.1 KIOPS
IOR-easy read	1.4 GB/s	8.85 GB/s
MD-easy stat	3.4 KIOPS	115.2 KIOPS
IOR-hard read	3.0 GB/s	3.9 GB/s
MD-hard stat	12.5 KIOPS	54.5 KIOPS
MD-easy delete	33.9 KIOPs	37.6 KIOPS
MD-hard read	4.8 KIOPS	18.8 KIOPS
MD-hard delete	13.2 KIOPS	12.5 KIOPS
Aggregate Bandwidth	1.1 GB/s	4.0 GB/s
Aggregate IOPS	7.4 KIOPS	28.9 KIOPS
Total Score	**2.85**	**10.79**

Table 5.3: IO-500 benchmark results

found before, breaking down in performance when a large number of files is involved. Overall Lustre shows some strengths but also some considerable weaknesses, while the NVMe SSD file system delivers a solid performance in all regions, only lacking behind in throughput-intensive situations.

As a next step, the following chapter will revisit the important findings of this chapter and affiliate them with the main scenarios that can be expected for HPC applications.

6 Conclusion

An extensive series of tests was conducted, mimicking three different common I/O behaviors with IOR, comparing the metadata performance with mdtest as well as testing and comparing the overall performance with a small run of the IO-500 benchmark.
The results of IOR showed multiple scenarios from which we can derive advantages of the NVMe SSD file system that should be considered for possible use-cases of the proposed secondary file system.
Revisiting the findings for the Checkpoint pattern in section 5.2.1, significant weaknesses of the Lustre File System were revealed for frequent writes of small files, especially when caching is not possible. Although the performance that IOR measured on Lustre in some cases could keep up with the NVMe SSDs when caching was utilized, other tests in that category suffered from large deviations in metadata performance (in particular for very small files below Mebibytes). The secondary NVMe SSD driven BeeGFS performed much better both with and without caching, offering a stable and quick performance. However, for a growing number of processes per node, its performance degraded significantly. Although it still clearly outperforms Lustre for 24 processes per node, more processes per node or larger files likely cause the BeeGFS to fall behind.
The outcome of the tests with the Append pattern illustrated this inclination, as the NVMe SSDs eventually reach their theoretical limitations and cannot keep up with Lustre's scalability in throughput-intensive cases. With 12 processes per node the NVMe SSDs peaked for file sizes of 512 MiB or more. Therefore, in situations with high concurrency and large files, Lustre is outperforming the NVMe SSDs. Nevertheless the results for smaller files and/or low process counts confirmed the in part significant advantage of the secondary file system in those situations.
Mdtest also exposed Lustre breaking down in performance when a high number of files is created in a short time span. This should be kept in mind for applications that for example pre-create a large number of files for later use.
In addition to these main scenarios that could be found in the results, there are a few more takeaways that were exposed by the results, including a rather troublesome minimal write time of almost 1 second on Lustre when caching was surpassed. This was the case for tiny file sizes of 32 Bytes up to hundreds of Megabytes, which is when the writing process eventually starts to take longer.
Furthermore Lustre encountered problems when using 24 processes per node and file-sync and/or checkWrite on files beyond a few Megabytes. This indicates some underlying issues on Lustre.
Another takeaway was the bad scaling of shared files. On both systems shared files

could not surpass around 2 GB/s even for large block sizes and a high number of processes. However, a possible reason is poorly optimized data striping, which typically can be fixed by the user itself.
Lastly the metadata tests showed somewhat disappointing results on the NVMe SSDs, not even approaching close proximity of their specified hardware limitations. This was also found by another study that specifically concentrated on metadata performance improvements through SSDs [**?**].
On the base of the conclusions presented in this chapter, the following chapter will give a recommendation on how the secondary file system would be best utilized as well as an outlook to the future.

7 Recommendation and Outlook

With the findings discussed in the previous chapter, an on-demand file system as proposed would be best fit for applications that include one or more of the following scenarios:

- metadata intensive
- works on (many) small files
- uses only few processes per node (to transfer data)
- cannot make use of caching
- regularly stores data during execution and abandons it afterwards

As each node that is involved should have its own NVMe SSD, the amount of processes per node that transfer data is paramount in determining whether this secondary file system would perform better than the main file system. If the number of processes transferring data on each node is too high or the files are too large, the NVMe SSDs will reach their hardware limitations and the main file system could provide better scalability. The limits of how large the files can be and how many processes per node still work better on the secondary file system are hard to determine. Table 5.2 can be taken as a very rough guideline for up to 48 processes (12 processes per node) when trying to answer this question for the special case of the Lustre file system on CLAIX.
Another important point is the big impact of caching on Lustre, in particular for small files. Applications that can not make full use of caching, for example due to high memory needs or necessary reads of files by processes on other nodes, would possibly perform better on the NVMe SSD file system.

The results showing the NVMe SSDs reaching their bandwidth limitations allow us to predict how other devices like conventional SATA SSDs with lower limitations would perform in the Append pattern test in place of the NVMe SSDs. A typical SATA SSD with around 500 MB/s write bandwidth instead of the NVMe SSDs would most likely already fall behind the tested Lustre File System for 256 MiB file sizes and 2 processes per node or 32 MiB file sizes and 12 processes per node. It most probably would reach its peak bandwidth already for even smaller files. This would severely limit the usefulness to a very small range of applications.

The presented criteria answers the main questions of this thesis, whether the proposed secondary file system can be useful and if so, in which situations specifically. However, there are still questions that need to be answered, as for example the problem of re-centralizing the data after using the secondary file system is not yet solved. To make the best use of the secondary file system, the impact of transferring data to the main file system after the application finishes should be examined. To keep this impact as minimal as possible, it would be preferable to leave little to no data that has to be transferred. This is common for some HPC applications, as they regularly store data during the execution (eg. checkpoints for restarts or matrices for calculations that are too large for the memory) and completely abandon it afterwards.
Furthermore, this thesis can at best offer vague predictions on how this secondary file system might perform on larger scale or different hardware. Once this supplement is implemented on more than a few nodes, further testing might be needed to confirm these predictions and provide reliable results to give the best possible advice to application developers on how to take advantage of the system.
Moreover, the secondary file system would allow a separate and more specific optimization for the use-cases suggested above. For example, previous studies have elaborated on software-side optimization to improve metadata performance [**?**] [**?**]. Although these optimization strategies are not available on Lustre, these kind of techniques might be worth further examination to tune the secondary file system to its fullest potential.