Bachelorarbeit

Evaluation of File Systems and I/O Optimization Techniques in High Performance Computing

Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften
Fachbereich Informatik

vorgelegt von

Christina Janssen

Email: janssen.christina@gmx.de
Studiengang Informatik
Matr.-Nr.: 5812740
Fachsemester: 6

Erstgutachter:	Prof. Dr. Thomas Ludwig
Zweitgutachter:	Prof. Dr. Norbert Ritter
Betreuer:	Michael Kuhn

Hamburg, December 5, 2011

Abstract

High performance computers are able to process huge datasets in a short period of time by allowing work to be done on many computer nodes concurrently. This workload often poses several challenges to the underlying storage devices. When possibly hundreds of clients from multiple nodes try to acces the same files, those storage devices become bottlenecks and are therefore a threat to performance. In order to make I/O as efficient as possible, it is important to make the best use out of the given resources in a system.

The I/O performance that can be achieved in a system results from a cooperation of several factors: the underlying file system, the interface that connects application and file system, and the implementation. Based on how well all of these factors work together, the best I/O performance can be achieved.

In this thesis, an overview will be given of how different file systems work, what access semantics and I/O interfaces there are and how a cooperation of these, in addition to the use of ideal I/O optimization techniques, can result in best possible performance.

Three I/O interfaces, POSIX I/O, MPI-I/O and ADIOS (the ADaptable I/O System), will be examined in more detail by measuring the performance of every single one in combination with two file systems, NFS and the parallel file system PVFS, in a real application. The results will show that ADIOS, especially in combination with PVFS, achieves best performance overall. However, it will also become clear that in order to achieve an ideal result, it is important for the interface to meet the application's requirements.

Contents

1 Introduction **1**

2 File Systems **4**
- 2.1 The General Design of Distributed File Systems 4
- 2.2 NFS 6
- 2.3 GPFS 10
- 2.4 Lustre 13

3 Semantics **16**
- 3.1 POSIX Semantics 16
- 3.2 Session Semantics 17
- 3.3 MPI-I/O Semantics 19

4 I/O Interfaces **21**
- 4.1 POSIX I/O 21
- 4.2 MPI-I/O 22
- 4.3 ADIOS 23

5 I/O Optimization Techniques **28**
- 5.1 POSIX I/O Extensions 28
- 5.2 MPI-I/O 29
  - 5.2.1 Implementation of ROMIO 33
  - 5.2.2 Data Sieving 34
  - 5.2.3 Two-Phase 36
- 5.3 File System Support 37
  - 5.3.1 MPI-I/O on Top of GPFS 39
- 5.4 LACIO 40
- 5.5 An Ideal Interface for HPC 42

6 Evaluation **45**
- 6.1 The Experiments 45

6.2 The Results 50

7 Conclusion and Future Work **57**

Bibliography **59**

Appendix **61**

A Application runtimes **61**

List of Figures **63**

List of Listings **64**

Chapter 1

Introduction

High Performance Computing (HPC) plays a big role in science today. Not only in the most obvious fields, such as physics and engineering, but also in medicine, climate research and other fields of scientific research, supercomputers have been on the rise for years. The Top 500 List, which is updated twice a year since 1993, gives an overview about the fastest supercomputers in the world. The current number one, Fujitsu's K computer, consists of more than 700.000 cores and reaches a speed of up to 10 petaflops[1] [Top]. When keeping in mind that the first computer was only built in the 1940s, 70 years ago, it is astonishing to think of what has been achieved in such a short period of time. Not only exclusively for scientific purposes, but even a major part of regular personal computers, often do not consist of a single core anymore. Multi-core processor systems, which basically are small parallel computers, are on the rise even for personal use.

With the increasing speed of processors, there is also an increase of data that has to be processed. However, if processors are able to calculate huge datasets in a short period of time, this does not mean that the transportation of this data from disk to CPU and back is just as fast. The contrary is the case. With increasing amounts of data to be transported the underlying storage devices may pose a bottleneck. In a high performance computer there are clients on hundreds of nodes, all of which might want to access the same data at the same time. Therefore, it is important to make the best use of the given resources in order to make I/O as efficient as possible. In this thesis, an overview will be provided over the resources that can be used in order to make I/O more efficient. The I/O performance that can be achieved in a system results from a cooperation of several factors: the underlying file system, the interface that connects application and file system, and the implementation.

[1] 1 flop = 1 floating point operation per second; 8 petaflops therefore are one quadrillion (thousand trillion) floating point operations per second.

Based on how well all of those factors work together, the best I/O performance can be achieved.

The next chapter will start by taking a closer look at file systems, paying special attention to the ones used in HPC. Some of the questions that will be answered are: What file systems are currently being used in supercomputers? And what makes these file systems so attractive to high performance computing? Three file systems will be presented in greater detail.
At first, the Network File System (NFS) will be introduced. It is probably one of the oldest file systems that is still used in some high performance computers today, and – strictly speaking – rather a network protocol than a file system. Furthermore, a closer look will be taken at GPFS, an IBM based file system, and Lustre, probably the most-used file system in HPC, which is also installed on the K computer.

Chapter three describes different file access semantics. There are strict access semantics, like POSIX semantics, which can hinder I/O, while there are other, more relaxed semantics, like session and MPI-I/O semantics, which are more prominent in HPC. A short overview will be given about all three semantics, as well as examples, how these semantics work, and in which way they influence I/O performance.

In the fourth chapter, attention is brought to I/O interfaces, which provide the connecting layer between the application and the file system. Again, three interfaces will be examined in more detail, starting with POSIX I/O, the standard interface of many file systems. It has been around for more than forty years and grants portability across many different systems. However, or maybe rather because of this, it also poses boundaries to HPC, which will be shown. The second interface to be inspected is MPI-I/O, which is part of the MPI-2 specification, extending it by an I/O chapter. MPI-I/O can be considered the quasi-standard in HPC applications today, as it provides rich functionality to improve I/O performance, the most significant probably being MPI datatypes, which allow noncontiguous file access.
A still relatively new interface is presented last in this chapter - the Adaptable I/O System, in short ADIOS. Different to other interfaces, ADIOS allows to easily switch between different I/O methods for the same application by simply modifying parameters in a separate XML file. Thus, clients are able to try different I/O methods for their applications and even provide hints to the file system without modifying the actual source code.
The three interfaces presented here will be taken up again later, in the evaluation part of this thesis, where their performance will be compared in a real application.

After giving an overview about the different file systems, file access semantics and I/O interfaces, chapter five focuses on I/O optimization techniques. It will be shown, how all of these three factors can efficiently work together in order to achieve the best possible performance. There are certain I/O techniques

that are specialized for certain file systems, like for example data shipping for GPFS, just as there are other techniques that only work when certain semantics are supported. In addition to giving an overview about I/O techniques, there will be thoughts about how the file system can help in improving performance and how an ideal interface for HPC might look like.

Chapter six contains an evaluation of the previous thoughts. The performance of different I/O interfaces – POSIX I/O, MPI-I/O and ADIOS – in combination with two file systems – the Network File System (NFS) and the Parallel Virtual File System (PVFS) – will be examined and compared by using `partdiff-par`, an application that is designed for solving partial differential equations. At the end of this chapter, there will be a section presenting the results and evaluation.

The last chapter presents concluding remarks and ideas for future work.

Chapter 2

File Systems

There are many different file systems available. Based on the way that data is stored in a file system, it can be differentiated between local file systems, where data is stored on one central device, and distributed file systems, where data is distributed across multiple nodes. The focus in this thesis will be on distributed file systems in a high performance environment. In this chapter, the general design of distributed file systems will be described first, being followed by a more detailed presentation of three different file systems. The focus will be on the differences between them, discussing advantages as well as disadvantages in the way they work and showing optimization potential.

2.1. The General Design of Distributed File Systems

In contrast to local file systems, in a distributed file system data is stored on multiple nodes and can be accessed by several clients in parallel. The general layout of a distributed file system is illustrated in figure 2.1. There are three different roles that nodes can play in a file system: I/O nodes, which store the data on disk, compute nodes, on which the applications run, and metadata nodes, which are responsible for administering any metadata. Depending on the kind of file system, nodes can play more than one role.

Generally (but depending on the file system, differences are possible), file access happens by the compute node contacting the metadata server, which gives out a file handle. The compute node then is able to access the file (using this file handle) by directly contacting the I/O node, which is responsible for the storage device that the requested file is stored on. In addition to this, metadata servers and I/O servers exchange information in order to guarantee

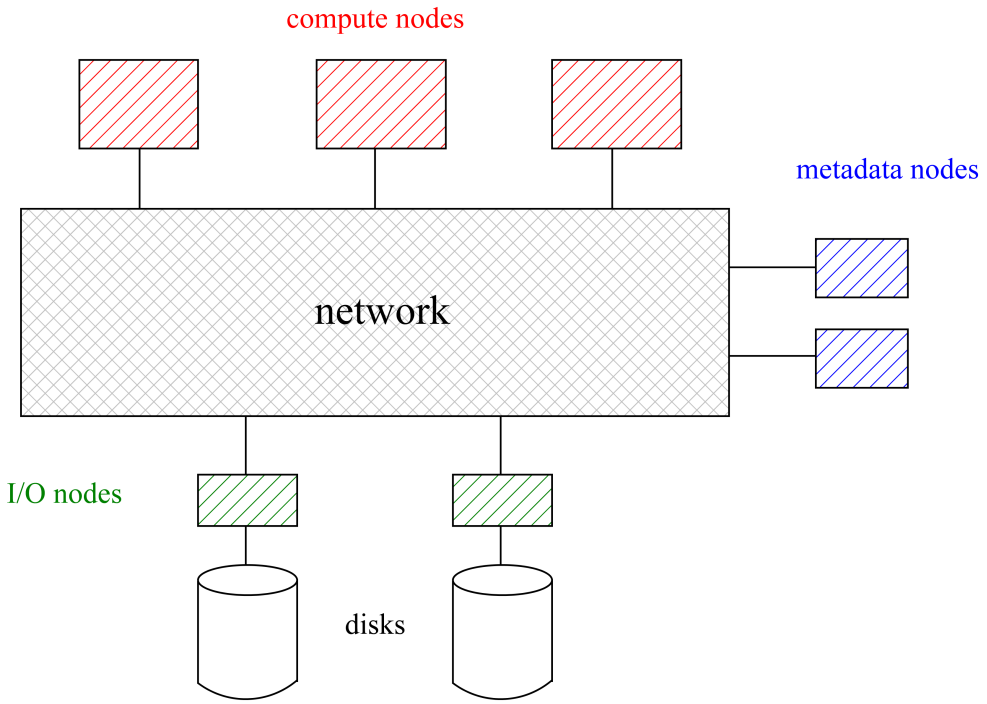

Figure 2.1.: General layout of a distributed file system

a consistent view on all files. Therefore, it is very likely that communication happens between all three kinds of nodes, which poses a challenge to the network.

However, if multiple client and concurrent file access is allowed, the most important challenge for a file system is to keep data consistent at all times. For this reason, it is important for a client to know, if there are consistency semantics specified in the file system. And if so, how is consistency achieved? Depending on its consistency semantics and methods, performance can be significantly decreased, for example if the file system achieves consistency by excessive locking.
A challenge going along with this, are the file system's caching semantics. Is caching allowed? And "where" does it happen? In a file system there are three locations possible for local copies: the server's memory, the client's memory, or another client's memory. It has to be made sure that the data in all three places is consistent and up to date.

In the following sections, these three file systems will be discussed in more detail, inter alia focusing on the challenges mentioned above:

- The Network File System (NFS), a protocol which allows data access over a network
- The General Parallel File System (GPFS), a symmetric cluster file system, and

- Lustre, an asymmetric, object-based cluster file system

In the process of deciding which file systems out of the variety of available systems should be presented here, several factors played a role. First of all, it was important to find file systems that do not have the exact same architecture. Many file systems that are used in high performance computing are similar in their design (as mentioned above). However, there are some remarkable differences in some of them. With GPFS and Lustre, two file systems have been chosen that are similar in their architecture, but at the same time provide striking differences in how this architecture is realised. GPFS is a symmetric system[1], whereas Lustre is asymmetric and uses an object-based approach. Compared to these two cluster file systems, NFS has a different design altogether. While being originally designed for single servers, it is still widely used in parallel computers today. It is interesting to see NFS's development over the last twenty-five years and to show up its differences to GPFS and Lustre, which were designed especially for the use on parallel, high performance computers.
Going along with this, another factor that played a role in the selection process was, how well are these systems represented in high performance computing today? As this thesis deals with file systems and I/O optimization techniques, it is important to know what file systems are used in HPC, and how well they perform. Lustre, being open source, which is another factor that certainly adds to its wide popularity, is used on the current top three supercomputers of the world and many others in the Top 500 List [Top]. GPFS is an IBM developed file system, which is used on high performance computers such as the ASC Purple Supercomputer [IBM]. NFS is the file system, which runs on the cluster that was used for the evaluation part of this thesis, and is therefore important to be examined properly.

2.2. NFS

The Network File System was initially designed for single machines, the main goals being portability to other operating systems and architectures, easy crash recovery and providing transparent access to remote files at reasonable performance (ideally comparable to local file access) [SGK+85]. With the first version originally having been developed by Sun Microsystems in 1984, it is one of the oldest file systems that is still used in high performance computing today (however, with considerable modifications compared to the original version).

NFS's major advantage is that it is independent of architecture and operating systems. By strictly separating the protocol and its implementation NFS

[1] In a symmetric system, in contrast to an asymmetric system, metadata is not stored centralized. This will be explained in more detail in 2.3.

can be used on different architectures and operating systems, which makes sharing resources easy even in heterogeneous environments. NFS (until version 4) actually uses two protocols: the 'mount protocol', which establishes the first connection between client and server, and the 'NFS protocol', which provides the client with the possible file operations, a set of Remote Procedure Calls (RPCs).

The general architecture of an NFS file system is illustrated in figure 2.2. The client connects to the server over a network via the 'mount protocol', mounting the file system. Transparency is provided by making the remote files seem as if they were local. After connecting to the server and mounting the system, operations on the files can be performed by calling RPCs.

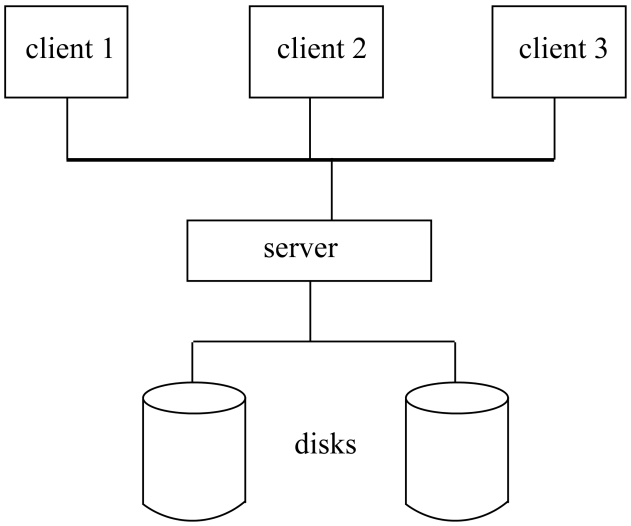

Figure 2.2.: The Network File System

In contrast to many other commonly used file systems, NFS (until version 4) uses a stateless protocol, which means that it does not store information about client requests, but all necessary information must be included in every request made to the file system. This means that recovery in the case of a server crash is comparatively simple. The client just repeats sending its request until the server is back and recovered and no information will be lost.
However, the stateless server does not have an overview about what files are being viewed and/or modified by which clients, which causes problems when files are being accessed concurrently. This did not pose a challenge when NFS was only used on single machines due to the atomicity of single machine operations. But with the rise of parallel computers, consistency guarantees for concurrent access have to be defined. Additionally, the server's statelessness increases the amount of requests from the clients, because the server does not store any information about a client's previous requests, so that all information has to be sent again if the file is needed again later. This may cause a bottleneck on the server side because of the amount and also the size of requests it has to deal with. Since it is stateless, with every modifying request the server has

to write back all modified data to stable storage before returning in order to prevent data loss.

This problem of the early version of NFS was soon apparent, so that client-side caching and file handles were introduced with NFSv3 in 1995 [PJS+94]. Making it possible for clients to have local copies to work on decreased the number of requests being made to the server, however new problems were rising: how to maintain file consistency between working copies?

NFS uses session semantics to provide cache consistencies. In contrast to POSIX semantics, which require that modifications to an open file must be visible immediately to other clients that have that file open, session semantics state that all changes being made to a file are not visible immediately to other clients but once the file is closed. Therefore, modifications are visible only in later sessions when the file is opened again.
For better performance, there are actually two client-side caches: one for file blocks and a second for file attributes. On any `open` request, cached attributes are being validated with the server. Only if any changes are detected, the client will update the requested file in its file block cache. However, this is not the perfect solution, as becomes clear when you look at files that have been modified by the client itself. In this case, the attributes in the client-side cache do not match with the attributes that are stored on server-side. Consequently, the client may invalidate the cache even though this is not necessary, which slows down the operation [PJS+94].

With NFSv3, another problem has been tackled: performance issues due to synchronous writes. In version 2, the protocol requested that all data that was modified by a client's `write` request had to be written back to stable storage synchronously by the server before returning from the request. This means that no requests by other clients could be served until the server returned from the request and finished writing the data back to stable storage. With NFSv3 it is possible to allow asynchronous writes for `write` and `commit` procedures [PJS+94]. In case of a server crash before data could be written back to disk asynchronously, this poses a risk for inconsistent data. However, to maintain consistency, a write verifier was introduced and served as a way for clients to detect any server crashes. The write verifier is an argument which is included in replies for `write` and `commit` requests. Usually it consists of the server boot time, which is unique due to the requirement that after every crash the server has to update its boot time. The client now is able to detect any server crashes by comparing the write verifiers of the `write` and `commit` requests. In case of a crash, the client has to repeat both requests, `write` and `commit`, in order to make sure that the changes are being written back to disk properly.

NFSv3 remained stateless in order to maintain fault tolerance and easy recovery [PJS+94]. However, this fault tolerance is achieved by simply resending requests to the server. This is acceptable for most of the requests, the so-called idempotent requests like `open` and `close`, which have the same result

if executed more than once by the server. There are, however, problems with non-idempotent requests, like for example `remove`. If a `remove` request is executed more than once, there will be an error message after the second request stating that the file to be removed does not even exist. Consequently, the results after first and second try are not the same. The stateless protocol, however, requires idempotency. In NFSv3 this problem was fixed by introducing a reply cache on the server, which stores recently serviced requests [PJS+94]. If a nonidempotent request has to be served more than once, the server will simply send the reply stored in the reply cache back to the client. Thus, turning nonidempotent into idempotent requests.

In NFSv4, which was presented in 2000, state was introduced to the file system [PSB+00]. In addition to stateful operations like `open` and `close`, file locking has been fully integrated into the protocol. `open` is an atomic operation for file lookup, creation and share reservation. The initial file handle for a file is retrieved by calling `open`. A share reservation can be issued to a client calling `open`, which denies read and write accesses of other clients to the file. `close` releases the state of the file issued by an `open`.
Next to share reservations, leases for lock management have been introduced, which provide a (server-specified) time-bound grant of control over the state of a file [PSB+00]. In order to keep a lease, the client is responsible for contacting the server and refreshing the lease. The use of leases avoids the risk of a permanent lock on a file, which cannot be revoked if the client that holds the lock on the file crashes and never recovers.

Another structural change that has been realised in NFSv4 is the introduction of the `compound` RPC procedure. In contrast to previous versions, in NFSv4 actions are no longer RPC procedures. Instead, work is done in *operations*, which form parts of `compound` procedures and correspond functionally to the RPC procedures in previous versions. With help of this `compound` procedure the client is able to group several operations into one request to the server. The server is in turn able to group the operations' replies into one single response. Therefore, as Pawlowski et al. state, the `compound` procedure is essentially the only RPC procedure left in NFS, together with `null`, and has been introduced in order to reduce network latency for related operations [PSB+00].
Error handling is kept simple. If an error occurs, the server sends a reply with all responses of operations that have been evaluated until the first occurance of the error. Consequently, when serving a `compound` procedure, replies are always sent either at the end of the last operation or if an error occurs [PSB+00]. For this reason, it is not recommended to group unrelated operations into one `compound` procedure, as evaluation stops when the first error occurs.

Overall, one can say that NFS has changed quite considerably since its first version from 1985, especially since introducing state in NFSv4. However, there are still some facts that are not satisfactory to the needs of high performance applications, such as metadata lookup and the overall single server design. In

the following two sections, two distributed file systems, GPFS and Lustre, will be presented, which were designed especially for high performance purposes and are used in some of the fastest supercomputers of the world.

2.3. GPFS

The General Parallel File System (GPFS) is an IBM file system for cluster computers. It was designed especially for high performance purposes, while at the same time maintaining the simple appearance and behaviour of a POSIX file system [SH02].
GPFS has a shared disk architecture, consisting of up to several thousand disks. By striping file data across all disks, the file system is able to achieve the aggregated throughput of all disks together without posing a bottleneck to the I/O system. This shared disk architecture allows for GPFS's extreme scalability.

The architecture of GPFS consists of three parts (see Figure 2.3): the compute nodes, the shared disks where the data is stored, and a switching fabric in between that connects the computing nodes with the disks. Instead of a storage area network as switching fabric, there might be some I/O server nodes instead, which provide access to the disks through a software layer running over a communication network [SH02].

GPFS is a symmetric file system, which means that metadata is distributed among nodes. Therefore, there are no dedicated metadata nodes, the same node can be responsible for metadata handling as well as serving I/O requests. Figure 2.3 shows the general layout of a GPFS file system (with I/O server nodes).

Files are striped across multiple disks, which is necessary in order to achieve high throughput. GPFS does not rely on a separate logical volume manager layer (LVM), but rather implements striping directly in the file system, which allows the file system to have enough control over fault tolerance and load balancing [SH02]. Striping is done in such a manner that files are divided into equal sized, rather big blocks (256 kB being the default size [SH02]) in order to allow large I/O requests and minimize overhead. Small files, however, are stored in smaller units, to keep seek overhead small.

In order to keep file data consistent on all nodes, GPFS has two approaches: distributed locking and centralized management. When distributed locking is used, every `read` or `write` operation has to acquire a read or write lock for the corresponding file portion, before reading or writing data, so that synchronization with conflicting accesses from other nodes to the same data is guaranteed. Whereas with a centralized management, all conflicting accesses are sent to one single node, which performs the requested operations. For

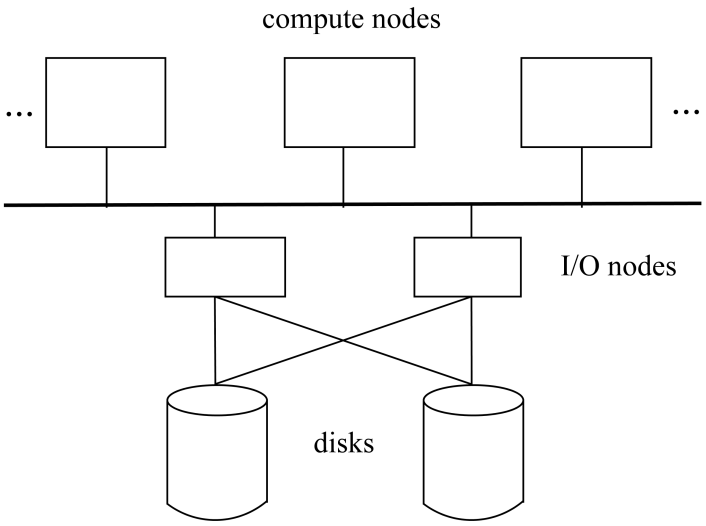

Figure 2.3.: GPFS Architecture

GPFS's architecture, distributed locking is ideal, as long as there are not too many overlapping write requests from different nodes, which increases locking overhead, as a lock has to be given out for every single `write` [SH02]. In this case, performance might be better if those conflicting requests are sent to a central node.

GPFS's distributed locking generally works like this: There is one centralized global lock manager on one node, and additionally there are local lock managers on each node of the file system. The global lock manager hands out lock tokens to the local lock managers on the nodes. With this lock token, the local lock managers are allowed to grant distributed locks to processes on their nodes. Consequently, if there is a request to access a file on one node, the local lock manager on that node passes a message to the global lock manager in order to retrieve the lock token for this file. If no other node owns a conflicting lock, the global lock manager hands out the token to the local lock manager. If a conflicting lock token is used by another node, the global lock manager has to retrieve it from that node first before it can be given out to the other node. For operations on the same node, once the local lock manager has obtained the lock token, no additional messages have to be passed in order to acquire a lock on that file. Therefore, distributed locking is one way of reducing locking overhead, namely by decreasing the messages that have to be passed in the system in order to obtain a lock.
Owning a lock token for a file also allows this node to cache data, as it is guaranteed that no other node can modify the data without retrieving the lock token first.

With respect to locking, an important fact to consider is to find the right lock granularity. If lock granularity is too big, conflicting locks are more likely, whereas too small locks often result in more lock requests and locking overhead. For this reason, GPFS provides several locking approaches, like for example

byte-range locking for updates to user data and centralized locking for managing file metadata [SH02].
A centralized approach to managing metadata is ideal, as metadata updates are often done from many nodes to the same file, which would otherwise result in extensive locking. For every node that writes to the same file, metadata has to be updated. In addition to this centralized approach, in order to keep locking overhead at a minimum, GPFS uses shared write locks on inodes, which store file attributes [SH02]. One node (out of every node accessing a file) is chosen as the metanode for the file, and therefore is the only node that writes the inode to or from disk. All other nodes keep a copy of the inode in their cache, which they update instead, and send their modifications to the metanode periodically. The metanode merges these modifications by taking the largest file size and latest `mtime` (modification time). When the shared write lock is revoked, which happens by a `stat` or `read` operation (if attempted to read past end-of-file) on another node, all updates have to be sent to the metanode immediately, as these operations require the exact file size and/or modification time.

As mentioned above, byte-range locking is used for requests to write file data. The smallest unit in I/O is one sector [SH02], so even if byte-range locking is used, a lock that is smaller than one sector cannot be given out. As has become clear, lock granularity plays a role in how severe locking overhead is. GPFS provides a possibility to decrease locking overhead if finer grained locks are needed. This is done by disabling byte-range locking and switching to data shipping mode [SH02] (also see section 5.3.1 for more information about data shipping). Here, file blocks are assigned to nodes, so that there is only one node that handles requests to a particular file block. Requests from other nodes to this file block are forwarded to this particular node. Studies by Schmuck et al. have shown that data shipping helps to increase performance for applications that share fine-grained data. This is due to the fact that fewer messages are exchanged than it would be during a token exchange. And also, flushing data to disk on a token exchange can be avoided [SH02]. However, this technique can only be used for applications that do not require POSIX semantics.

Concluding, one can say that GPFS is specified for and provides techniques to improve performance for high performance applications. Compared to NFS, GPFS provides different locking techniques, out of which the most fitting can be chosen for each operation, like for example distributed locking for modifying file data and a centralized approach to manage metadata updates. A special focus in improving performance has been given to metadata handling, as this is an important factor to consider, especially in high performance computing, where a lot of data is processed and file updates happen frequently. GPFS has found a way to keep locking overhead, especially for metadata updates, at a minimum.

2.4. Lustre

Currently being used on 15 of the world's top 30 supercomputers, including the number one of the Top 500 List (of November 2011), Fujitsu's K Computer, Lustre is the dominating file system on high performance computers [Top].

In its architecture, Lustre is similar to GPFS (see figure 2.4), with the exception that Lustre is an asymmetrical file system, which means that metadata is centralized. Nodes cannot have several roles, so that a metadata server can not be an object storage server at the same time. There are five components in a Lustre file system: metadata servers (MDS), metadata targets (MDT), object storage servers (OSS), object storage targets (OST) and the compute nodes. The roles that these components play in performing I/O is described in the following: MDSs handle requests for data on the MDTs, which store the files' metadata (such as filenames, permissions and the file layout). There are usually one or more MDTs attached to every MDS. OSSs provide file I/O service (usually to several OSTs, where the file data is stored in one or more objects), while direct communication between client and OSS is possible.
Here is an example for a simple I/O routine: When a client opens a file, a request is passed to the MDS, which retrieves a set of object pointers and their layout from the MDT and sends them back to the client. The client then is able to perform I/O by directly interacting with the OSS nodes, which store references to the objects on the OSTs. By separating the file I/O (between client and OSS) from the metadata I/O (between client and MDS), performance can be improved.

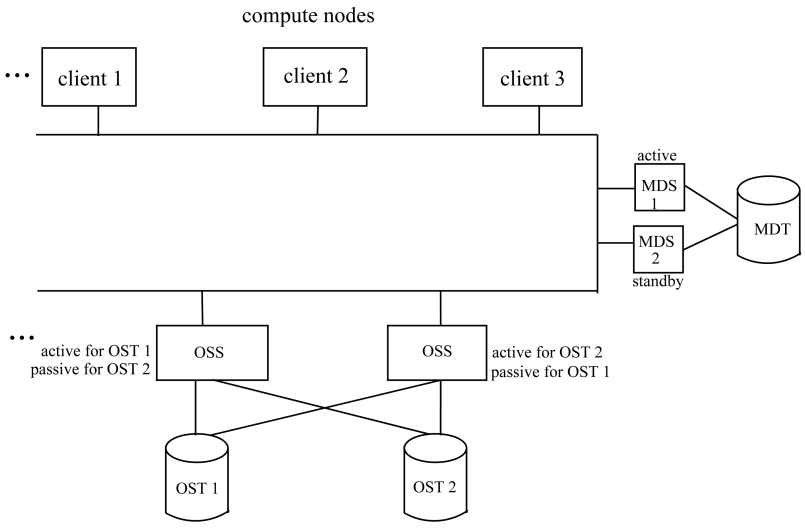

Figure 2.4.: Lustre Architecture

When looking at the roles that the single components play, another contrast to GPFS is striking: Lustre is an object-based file system. This means that the storage servers do not store references to the data itself, which is stored on the disks (as in GPFS, for example), but rather hold references to objects on

the OSTs, which are implemented as files on the OSTs [Inc02]. Each of such an object is associated with an MDT inode, which holds information about the file.

Similar to GPFS, files are usually striped across several disks in order to profit from the aggregated I/O bandwidth. When striping is used, MDT inodes hold references to more than one object of the same file, so that the file is striped across several objects (each object being stored on separate OSTs). The striping layout (number of stripes, stripe size and OSTs that are used) can be configured for every file, which makes it possible to tune performance for every file individually [Inc02]. Each file can be striped across as many as 160 objects, with each object storing up to two terabytes, so that the maximum file size can be as big as 320 terabytes [Inc02].

Lustre uses distributed locking, guaranteeing complete cache coherency [Inc02]. Metadata locks are managed by the MDT that stores the inode for the file, while the MDS manages modifications to the inodes and is the only node that is allowed write locks for the inodes, similar to the metanode concept in GPFS. File data is managed by the OST, where the file (or the part of the file that is requested) is stored on. Byte-range locking is used, allowing overlapping read requests and non-overlapping write requests [Inc02]. Thus, multiple concurrent read requests for the same file can be granted, as well as concurrent write requests to different regions of the same file, reducing locking overhead to a minimum.

In addition to high performance I/O, Lustre also provides high availability and failover techniques. This is accomplished by replicating hardware in such way that in the case of a failure, a standby server can be used in the place of the failed server. There are two failover configurations possible for nodes: *active/passive* and *active/active* (see figure 2.4 for an example).
In the active/passive case, there is one active node, which provides resources and serves data, while the passive node stands by idle. If the active node fails, the passive node is activated and fills in for the failed node. This configuration is typically used for MDSs. In order to avoid completely idle nodes, the standby MDS often serves as an active MDS for another Lustre file system [Inc02].
In the active/active case, which is usually used for OSSs, both nodes are active. Each node provides a subset of resources, and in the case of a failure is able to take over the resources from the failed node [Inc02].
This failover concepts provides transparency, as clients usually do not recognize a failure in the system. The request to a failed node is automatically resent to the standby node if the original node does not respond within a certain time span. Thus, making the delay the only possible sign for a client to detect a possible failure.

It can be concluded that Lustre is very similar to GPFS in its architecture and locking mechanisms. However, one of the major differences lies in Lustre's object-based approach, which is supposed to improve performance by linking

requests to references of objects rather than the data itself. Additionally, Lustre has some simple, but highly effective failover techniques, which provide clients with a system that is highly available.

Summary

In this chapter, it has become clear that different file systems provide different advantages and disadvantages for use in HPC. NFS for example has changed quite considerably since its first version, however it does not meet the needs of high performance applications due to its overall single server design. In contrast to NFS, GPFS was designed specifically for HPC and provides techniques to improve the performance of high performance applications, for example by using both, distributed locking and a centralized approach in order to keep locking overhead at a minimum. While Lustre is similar to GPFS, it follows an object-based approach and provides highly effective failover techniques.

Chapter 3

Semantics

A file system's performance is strongly influenced by the interface's access semantics. While many parallel file systems are highly specialized in order to provide high performance to a certain recurrent access pattern, this specialization poses a bottleneck. On the other hand, there are file systems, e.g. NFS, which provide heterogeneous data access, allowing file access to a variety of applications, but at the same time limiting performance by their single server design. Access semantics define rules of how a file system reacts to file access demands and provides answers to questions such as: is concurrent file access allowed? And if so, how is consistency managed? When are modifications to a file visible to other clients? In order to achieve the best performance, an interface between application and the underlying file system is needed, which provides heterogeneous access and chooses the right access semantics and optimization techniques for the particular application and file system.

3.1. POSIX Semantics

"The majority of file systems support POSIX" [HNH09], a standard interface with a strict set of semantics. However, POSIX is not the ideal choice for high performance applications. The major point of POSIX semantics is that modifications to a file must be immediately visible to all other processes. This is extremely difficult to realize in a multi-node file system, as this will result in severe inter-node communication and locking overhead in order to achieve consistency. Locking granularity must be fine enough in order to achieve any parallelism at all. Otherwise, extensive locking will result in sequential file access, just like on a single server machine. If local copies are allowed, the local caches of processes that have a modified file in its cache have to be updated every time a change occurs. In a high performance environment with possibly

hundreds or thousands of computer nodes accessing files concurrently, this poses a major threat to performance. It is important to note that the POSIX API was initially developed for use on single server machines with sequential data access and thus only allowed synchronous reads and writes for a long time (compare NFSv2 and 3). There is a concept for nonblocking access, however, it has to be clearly defined what happens in case of a server crash if modified data is not completely written back to disk yet. In such a case, the client has to be notified and must send the data again. At the same time, reads by other clients to the modified file, have to either return all or none of the modified data in order to comply to POSIX semantics.

An example for POSIX semantics is presented in figure 3.1. There are two clients performing I/O on the same file data. Client 1 writes some bytes into a file, which is stored in its cache. Client 2 then reads the same file. As modifications have to be visible to other processes immediately (including an update to the server cache), client 2 reads the newly written data. Consequently, both clients now have the same view on the file.

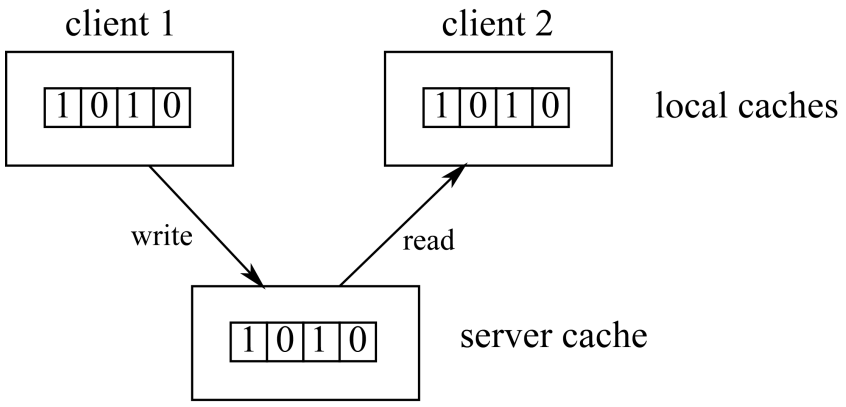

Figure 3.1.: An example for POSIX semantics

3.2. Session Semantics

A more relaxed alternative to strict POSIX semantics are session or "close-to-open" semantics, which are also implemented in NFS. As the name suggests, clients can access files in sessions, which means that they receive a local copy of the file to store in their local cache. Any changes being made to the file (file data or metadata) do not have to be visible immediately to other clients. Instead, all changes to the file are written back on the server when the client closes the file. Other clients are notified of these changes once they try to open the file. Therefore, session semantics are more relaxed than POSIX semantics, but still pose a potential threat to performance, as data has to be revalidated

every time a client opens a file, and also to consistency if cache coherency is not managed carefully.

In the following two graphics, there's an example how session semantics can lead to inconsistent file views. Similar to figure 3.1, two clients perform I/O operations on the same file data. In figure 3.2, client 1 writes some bytes to the file. Client 2 reads the data from the file. Since modifications are not visible to others before client 1 closes the file, client 2 receives outdated data from the server's cache. In figure 3.3, client 1 closes the file. Consequently, other clients and the server are updated about the modifications. Therefore, client 2 receives different data in its second read operation, even though there were actually no changes being made to the file between client 2's first and second reads.

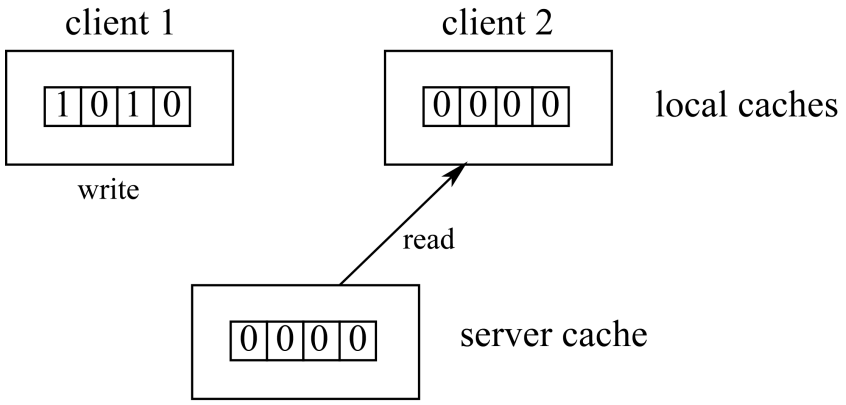

Figure 3.2.: Session semantics: client 1 writes some bytes to a file, client 2 reads outdated data

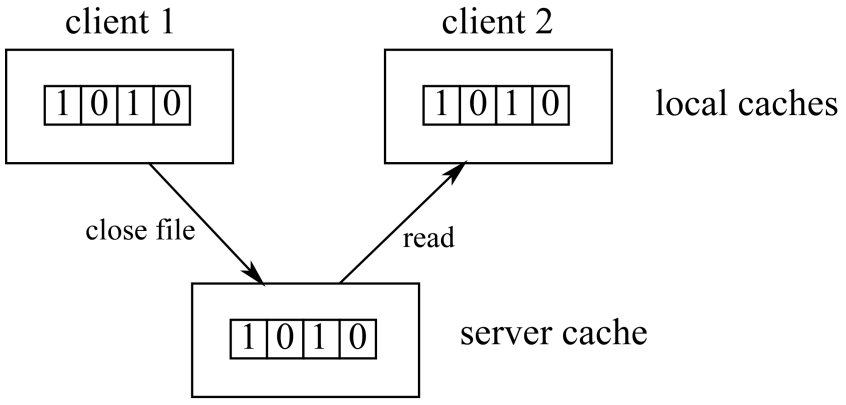

Figure 3.3.: Session semantics: modifications to a file are not visible to others before the file is closed

3.3. MPI-I/O Semantics

In contrast to POSIX, MPI-I/O semantics make it possible to achieve efficient parallel access for several clients on the same file. MPI defines an interface for message passing operations between processes. With MPI-2, an I/O chapter was added to MPI, the so-called MPI-I/O, which provides a set of I/O operations. Major goals here are portability and high performance. In addition to the possibility of allowing applications to manage cache coherency themselves "by controlling when data and metadata are flushed and revalidated" [HNH09], MPI-I/O supports noncontiguous file access, asynchronous I/O as well as collective operations. Code-level portability across architectures and operating systems is provided, as well as opportunities for scalable versions of `open`, `close` etc. MPI-I/O works in a similar way as message passing between processes.
Consistency is achieved by the Sync/Barrier/Sync construct, which is performed by all nodes between write and read requests, so that modified data is visible on all nodes. This is performed in three steps [HNH09]:

- SYNC: modified data is placed on filing servers
- BARRIER: wait until all modified data is written to the servers, while ensuring that no read requests are issued until all clients have the same view of file contents
- SYNC: file revalidation by making sure that written data is visible to all nodes

It can be said that MPI-I/O's consistency semantics are weaker than POSIX's. Different to POSIX, a write from a process is not immediately visible to other processes, but only to processes in the same communicator. In order for any modifications to be visible to other processes not in the same communicator, both parties, the reader and writer, have to call the `MPI_File_sync` function [TGL99b].
As a result, on file systems that implement POSIX consistency semantics (e.g. Lustre), MPI-I/O's consistency semantics are always automatically supported. An example contrary to this would be NFS, which does not implement POSIX but session semantics and therefore encourages noncoherent client-side caching. In order to achieve consistency when using MPI-I/O on NFS, it is necessary to use byte-range locking for reads and writes, as ROMIO does, so that local copies of data in the client's cache are avoided [TGL99b].

Byte-range locking is also used in order to achieve atomicity for noncontiguous MPI-I/O requests. This is necessary to avoid processes from writing concurrently into overlapping file regions. In MPI-I/O, two atomicity modes are possible: nonatomic, which is the default mode, and atomic. POSIX only supports atomic mode, so this is the default mode on file systems that imple-

ment POSIX semantics and it guarantees atomicity for contiguous MPI-I/O requests. For noncontiguous MPI-I/O requests, however, atomicity cannot be guaranteed, if more than one write requests are made. For these cases, byte-range locking is used in order to lock the necessary section in the file, which is requested for I/O [TGL99b].

Summary

In this chapter, it has become clear how access semantics may have an impact on I/O performance. In order to achieve the best performance, it has to be decided which semantics work best in the individual case. If strict data consistency can be neglected, for example if it is clear that there will not be any concurrent, overlapping write modifications, strict semantics can be relaxed, which may lead to better performance. In the following chapter, three I/O interfaces will be presented, and the differences will become clear in how each interface implements different access semantics.

Chapter 4

I/O Interfaces

The interface, serving as a connecting layer between the application and file system, plays a significant role in how well an application can perform. In this chapter, three interfaces will be examined in more detail: POSIX I/O, the defacto standard interface for file systems, MPI-I/O, which is part of the MPI-2 specification and was designed for high performance parallel I/O, and ADIOS, an interface which gives users the possibility to separate I/O handling from the actual source code and achieves best performance, which will be shown in a later chapter.

4.1. POSIX I/O

POSIX I/O is the standard I/O interface for many file systems and has been around for more than forty years [GWRG06]. Initially being designed for single machines with single memory spaces, it provides a portable interface with simple, consistent and powerful semantics, which are relatively simple to implement [GWRG06]. With the rise of high performance computing and its parallel applications, however, the POSIX I/O interface poses a boundary to high performance.
The two main problems that POSIX I/O poses to parallel applications are: first, the stream of bytes method used for file access, and second, the synchronization techniques used.

File access in POSIX I/O happens contiguously, when opening a file the process sees a flat layout of the address space for the file and a file pointer, which indicates the starting point for any I/O operation. I/O is stopped when the ending point for this operation is reached (a specified length (byte-range) from starting point). In this stream of bytes access model, no other unit than

the byte is known or could be used (like for example some form of MPI derived datatype). Additionally, there is no way of passing more information about the access pattern on to the file system (like for example hints in MPI-I/O), so that file access could be optimized.
Following POSIX semantics, modifications to a file are always visible immediately to other processes, so that metadata about the file has to be constantly updated. To ensure this, the file's time stamp is updated after every `read` and `write` operation, as well as the size of the file in bytes after every `write` operation. This is information that is not often needed for every process in parallel applications [GWRG06] and is therefore not necessary to be known to every process after every I/O operation.

In order to prevent multiple processes from accessing the same data in a file at the same time, synchronization techniques have to be defined. Unordered accesses have to be made ordered in order to fulfill POSIX semantics. These semantics do allow multiple writes to the same location in the same address space, however, only the last write is defined as the new content [GWRG06]. In order to implement these semantics, synchronization primitives [GWRG06] such as locking regions of the file or barriers are used. These kinds of synchronization techniques are relatively simple to implement and in addition to this, they are very fast on single machines with shared memory.

This changes, however, if synchronization techniques, such as locking, are used on parallel computers with multiple nodes and distributed memory. In this case, information about locking has to be sent over a network, which slows down the process considerably. As an example, Grider et al. have stated that locking a single value by the `test and set` operation only takes a few tens of nanoseconds, whereas the same operation on a parallel computer takes as long as a few tens of microseconds over the network on just one node [GWRG06].

Summing up, it can be said that parallel file systems that are based on strict POSIX semantics have to accept deficits either in performance or in data consistency. To tackle this issue, a working group of HPC users, the High End Computing (HEC) Extensions Working Group, has collaborated in order to propose goals for extending the POSIX I/O interface, so that it could be improved and achieve better performance even on parallel, high performance applications. More about this in the next chapter.

4.2. MPI-I/O

MPI-I/O is a part of the MPI-2 specification. The interface was designed "specifically for portable, high performance parallel I/O" [TGL99a]. Prior to this, there had been a lack of a portable interface for parallel I/O. As stated above, the POSIX API, which was used with slight variations by most parallel file systems [TGL99b], is not an ideal solution for parallel I/O. As part of the

MPI 2 standard, the MPI Forum defined a new API for parallel I/O: MPI-I/O, which was designed to achieve both, portability and high performance for parallel applications.

How does MPI-I/O achieve its portability? Since the UNIX functions (`open`, `write`, `close` etc.) are portable themselves, one possibility would be to simply implement MPI-I/O on top of these functions [TGL99b], but which would limit both, functionality and performance. Simply implementing MPI-I/O functions on top of the basic Unix-I/O functions poses several problems, the biggest probably being that the basic Unix-I/O functions are insufficient, so that many file systems provide additional functions, which are not portable from one to the other. Another problem is file consistency. MPI-I/O cannot handle local caches. For example, if used with NFS it is necessary to use locks in order to disable client-side caching, which might otherwise cause inconsistencies when several processes access the same file.

Implementing MPI-I/O functions on top of the POSIX I/O interface, which might be an alternative to consider, poses similar problems. It does provide more functionality than basic Unix functions, such as asynchronous I/O, but in spite of this, the POSIX standard is not supported on all file systems, at least not completely. On most file systems, only part of it is implemented. In addition to this, POSIX does not support some of the features MPI-I/O does, like data shipping for example, which is used in GPFS [TGL99b].

According to Thakur et al. "the only way to implement MPI-I/O portably with complete functionality and high performance is to have a mechanism that can utilize the special features and functions of each file system" [TGL98]. For this purpose, they developed ADIO, an abstract-device interface of MPI-I/O, which is also used in ROMIO, a portable implementation of MPI-I/O. ROMIO consists of two parts: the portable MPI-I/O implementation and the internal layer ADIO, the underlying interface which provides optimizations for different file systems (see Figure 4.1). ADIO is an additional layer in MPI-I/O. It provides portability to MPI-I/O across several different file system implementations.

A closer look at MPI-I/O's performance optimization potential and the implementation provided by ROMIO will be taken in the next chapter.

4.3. ADIOS

The I/O performance that applications can achieve depends on the I/O routines being used, whose performance again are dependent on several factors, such as: I/O patterns, I/O system and the compute system [LKS+08]. The ADaptable I/O System (ADIOS) is an API, which was designed in order to give developers a chance to experiment with different I/O techniques in a very

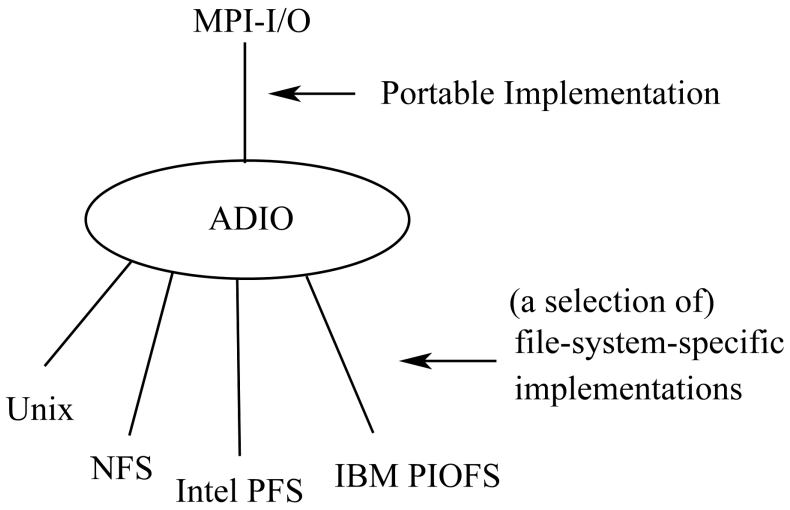

Figure 4.1.: ADIO design

simple way to find the most efficient way of coupling their application with I/O [LKS+08]. This is done by separating the actual source code from an XML configuration file, which can be used to make changes to the I/O configuration. This way, making changes to the I/O techniques is possible without touching the source code at all. At the same time, ADIOS provides fast I/O with an interface that is almost as simple as POSIX I/O [LKS+08]. Some factors which sparked the idea for such an API are described in the following.

In addition to the fact that I/O performance strongly depends on the I/O routines (such as MPI-I/O and POSIX I/O), it is important to note that I/O routines that work well on one system with a certain file system do not necessarily have to perform well on a different system with a different underlying file system. Also keeping in mind that different I/O patterns might be used on different parts of the code, it is important to provide a way to modify these I/O operations independently in order to get the best performance out of the application. This way should be kept as simple as possible, as it has to be able to deal with large-scale code that is performed on a high number of compute nodes, which makes I/O tuning complicated for the developer. Therefore, with ADIOS several available I/O methods can be tested for an application and measured for performance, so that the method with the best performance, while at the same time offering the required features, can be chosen.

While developing ADIOS, five main requirements were addressed [LKS+08]:

- Multiple, independently controlled I/O settings
- Data items must be optional - for example it should be possible for a main process of a group to write header information while the other participating processes do not

- Dynamic array sizes - have to be specified at run time in a way consistent with the standard I/O API
- Handling of reused buffers - support constructed output values in buffers
- Limiting buffer space for I/O - as scientific codes have strict limits on how much memory space is allowed to use for buffering

As can be seen in Figure 4.2 the ADIOS architecture consists of four parts [LKS+08]:

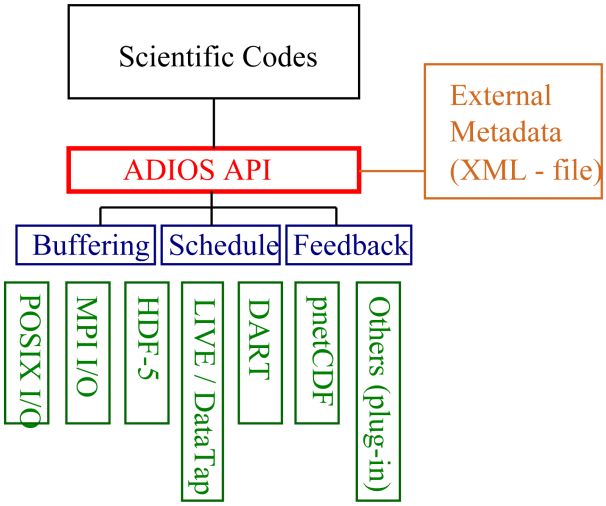

Figure 4.2.: ADIOS architecture

First, there is the ADIOS API, which provides the calls that are used in the scientific code (there is both, a Fortran and a C interface available). These calls can be grouped into two groups of operations:

- the setup/cleanup/main loop calls:
  `adios_init("config.xml")`, `adios_begin_calculation()`, `adios_end_calculation()`, `adios_end_iteration`
  These are global calls and perform the initialization and cleanup operations. Additionally, they give indications for when asynchronous methods and I/O operations should be used (namely outside of the begin and end calculation calls).
- the I/O operations:
  `adios_get_group`, `adios_open`, `adios_write`, `adios_read`, `adios_close`
  With `adios_get_group` a handle for a data group is retrieved, which can be used for opening the storage name. ADIOS supports both, synchronous and asynchronous operations, so supplied buffers have to be valid until `adios_close` is called, which functions as a 'commit' operation.

Second, there is a layer which provides common, internal services (such as buffering, encoding and decoding), which can be used by the transport methods.

Third, there is another layer providing the different transport methods. Supported methods are: synchronous MPI-I/O, collective MPI-I/O, POSIX, asynchronous I/O, and a NULL method for no output, used for benchmarking without any I/O. Asynchronous MPI-I/O is under development [LKS+08].

And fourth, there is an external XML file, containing elements and attributes by which configurations for the I/O can be done. There is for example a "method" attribute, which is a simple string that controls which transport method should be used. In this way, changing the I/O method for an application can be done by simply changing this attribute in the XML file.

Listing 4.1 is an example for a simple XML file, taken from the application that will be used later on in the evaluation part of this thesis.

Listing 4.1: Example XML file for ADIOS

```
1 <adios-config host-language="C">
2
3   <adios-group name="checkpoint" coordination-
       communicator="comm">
4   <var name="matrix" type="double"/>
5   </adios-group>
6   <method group="checkpoint" method="MPI"/>
7
8   <buffer size-MB="30" allocate-time="now"/>
9 </adios-config>
```

The language of the application's source code is specified by `host-language`. A group is defined by `adios-group name`. In this example the group's name is "checkpoint", as this group of processes writes a backup checkpoint every few iterations. The communicator is specified by `coordination-communicator`. For every group in the XML file, variables can be defined by choosing a name and a type. Variables are used for the read and write operations. In this example "matrix" is the buffer which contains the elements (doubles) that are supposed to be written to a file.

The transportation method, which should be used, can be issued by the `method` parameter. A method has to be chosen for every group defined in the XML file, but different methods can be used for different groups. Thus, it is possible to run the same application using different transport methods for I/O operations in the same run. Methods that can be used are, among others, POSIX, MPI-I/O and NULL. When issued NULL, the assigned group will be commented out, which makes it possible to run the application without I/O, which can be useful, for example in order to find out how much of the runtime

is needed for I/O alone.
The last parameter allocates a buffer of the size provided. The allocation time can either be "now", when the buffer is allocated immediately, or "on-call", meaning that the buffer will be allocated when ADIOS receives the information that all memory needed for calculation has been allocated.

A closer look at ADIOS will be taken in the evaluation part of this thesis (chapter 6), when ADIOS is one of the interfaces being used to examine I/O performance.

Summary

In this chapter, an overview has been given about three I/O interfaces: POSIX I/O, MPI-I/O and ADIOS. As has become clear, the I/O interface plays an important role when measuring the performance of an application, as this is the connecting layer between file system and accessing application. While POSIX has been around the longest, it was initially designed for local file system and is therefore not an ideal solution for HPC. MPI-I/O is more fitting for use in HPC, as it was specifically designed for HPC parallel I/O, while its implementation ROMIO provides portability to different file systems. ADIOS's approach of separating I/O configuration from the application's source code is an interesting innovation, especially as this allows easy modification of I/O routines in order to fit to a variety of file access patterns. How well this performs in a real application will be examined in chapter 6.

Chapter 5

I/O Optimization Techniques

After having presented different file systems, access semantics and I/O interfaces, information will now be provided about techniques which are used in order to improve I/O performance. The first major point will be the POSIX extensions, which were suggested by the HEC Extension Working Group in order to improve the POSIX I/O interface for use in high performance computing. Afterwards, MPI-I/O will be tackled, mentioning MPI derived datatypes as well as collective I/O. While presenting ROMIO, an implementation of MPI-I/O, information will be provided about further optimization techniques, such as data sieving and Two-Phase. Furthermore, the impact of file systems will come into play once again. As an example, it will be shown how GPFS can reach best performance while specializing on MPI-I/O. An alternative is presented with LACIO, a still relatively new technique, which (different to other techniques) keeps the physical data layout of files on disk level in mind, when performing I/O. The chapter ends with suggestions about how an ideal interface for HPC might look like.

5.1. POSIX I/O Extensions

There are three patterns which are often present in high performance computing: concurrent file accesses, noncontiguous file accesses, and high metadata rates [VLR+08]. The HEC Extension Working Group suggests extensions to the POSIX I/O Interface for every one of those patterns, as can be seen below.

- Concurrent file accesses
  The problem with concurrent file accesses, next to name-space traversal, are the synchronization techniques that have to be used in order to guarantee file consistency. When using the basic POSIX I/O operations,

this is often done by locking or the use of barriers, which slows down the application. To improve concurrent accesses, the use of shared file descriptors and group opens (`openg`, `openfh`) have been suggested. The `openg` call opens a file and returns a group handle, which corresponds to a file descriptor and can be passed to cooperative processes. With this group handle, the `openfh` function can be called, which will return the corresponding file descriptor. This file descriptor can be used in order to perform I/O on the file. The lifetime, scope and security of the group handle is up to the implementation.

- Noncontiguous file access
  Noncontiguous file access with POSIX I/O is already possible with the POSIX `listio` interface, which can submit mutliple I/O requests with one function call. There is, however, no reordering or aggregation possible in order to further optimize I/O. Furthermore, `listio` poses the restriction that the specified sizes in the memory vector and the file regions are the same, as well as the number of elements in the memory and file vector.
  Two new read and write functions are suggested: `readx` and `writex`. In contrast to the existing `readv` and `writev` functions, which specify a memory vector and perform serial I/O, `readx` and `writex` also read and write strided vectors of memory. Compared to the `listio` interface, requests made by `readv` and `writev` can be reordered by the implementation in any way that optimizes I/O.

- Metadata
  As mentioned above, some of the metadata which is provided by the `stat` call are time consuming to look up and might not be necessary in every call, like for example a file's last update time and/or file size. In order to make metadata lookup more efficiently, alternatives to the regular `stat` call have been proposed: `statlite`, `lstatlite` and `fstatlite`. Those functions return the same attributes as the `stat` call, but provide a litemask field, which can be used in order to specify which attributes are required to be valid. For an attribute that is not required to be valid, an outdated value might be returned.

5.2. MPI-I/O

As Thakur et al. have mentioned in their work, a large part of parallel applications make many small I/O requests to access noncontiguous pieces of data from a file [TGL99a], even though parallel file systems have been optimized to handle large requests. This problem results from general access patterns of parallel applications: many processes have to access many small chunks of data that are not located contiguously in the file, as well as problems of the

(in most cases) Unix-like interface that only allows to access a single, contiguous chunk of data in one request [TGL98]. In contrast to the POSIX API, MPI-I/O makes it possible to access noncontiguous data from files. Instead of making several small contiguous file access requests, MPI-I/O provides a possibility to sum these requests up into one large function call. To achieve this, the user has to define file views for each process and use collective operations.

MPI Derived Datatypes and Collective I/O

There are two kinds of datatypes in MPI-I/O: basic, like integers and floating point numbers, and derived. Derived datatypes consist of multiple basic and/or other derived datatypes, which can either be located contiguously or noncontiguously. By defining a derived datatype that describes the noncontiguous access pattern to a file, combined with defining a file view for every process accessing the file, the performance for I/O requests can be optimized. Thus, it can be avoided to make several small requests by using MPI-I/O's derived datatypes and collective operations.

Opening a file in MPI-I/O is a collective function, which means that every process in the communicator opens this file. Every process defines a special file view, which starts at a certain displacement (in bytes) from the beginning of the file. The local file view for one process consists of a sequence of the same filetype, which may be a basic datatype or a derived datatype (consisting of an elementary datatype (etype) and holes, (see figure 5.1)).

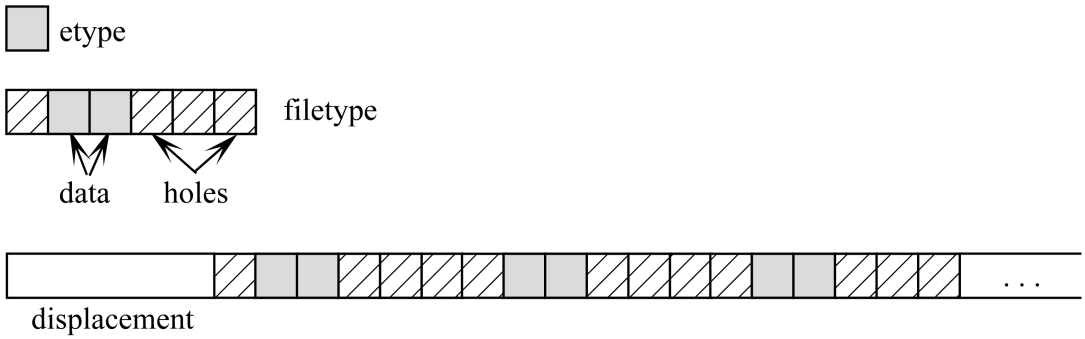

Figure 5.1.: Local file view of a process. Data that is visible to the process is marked in grey.

The filetype for each process defines what data in the file the process is allowed to access (since he only "sees" that particular data) and consists of the etype, filetype and displacement. The global file view for all processes consequently consists of the filetypes of the processes (see figure 5.2).

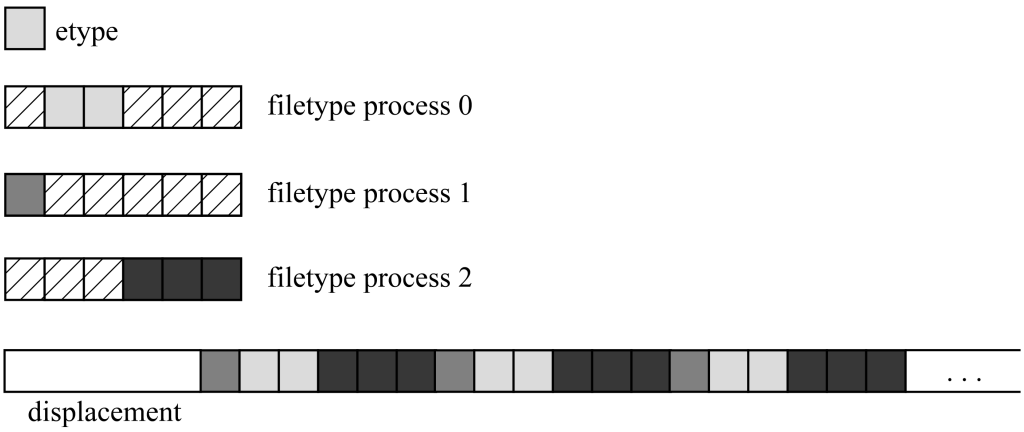

Figure 5.2.: Global file view of all processes. Every process only sees its sections of the file defined by its fileview.

Here an overview of the five basic collective I/O operations:

```
int MPI_File_open(MPI_Comm comm, char *filename, int
   amode, MPI_Info info, MPI_File *fh)
```

`MPI_File_open` is a collective operation, performed by all processes in the defined communicator with the same arguments. The file is specified by the `filename` and a file handle (`fh`), which is issued to every process opening the file. The parameter `amode` specifies the access mode, and `info` can be used in order to pass hints about the access pattern to the implementation, so that I/O can be improved by using this information.

```
int MPI_File_set_view(MPI_File fh, MPI_Offset disp,
   MPI_Datatype etype, MPI_Datatype filetype, char *
   datarep, MPI_Info info)
```

`MPI_File_set_view` is another collective operation, which defines the file view. The file handle has to be provided, as well as an offset (`disp`). `Etype` and `filetype` have to be specified, as well as the argument `datarep`, which specifies the file representation (`native, internal or external32`). `info` is of the same use as in the `MPI_File_open` function.

```
int MPI_File_read_all(MPI_File fh, void *buf, int count,
   MPI_Datatype datatype, MPI_Status *status)
```

`MPI_File_read_all` is the collective eqivalent to the non-collective function `MPI_File_read`. It requires a filehandle (`fh`) for the file that will be read from and a pointer to a buffer (`buf`), where the data will be stored. The arguments `count` and `datatype` provide information about the number and datatype of the elements to be read.

```
int MPI_File_write_all(MPI_File fh, void *buf, int count,
    MPI_Datatype datatype, MPI_Status *status)
```

`MPI_File_write_all` is the collective eqivalent to the non-collective function `MPI_File_write` and uses the same arguments as `MPI_File_read_all`.

```
int MPI_File_close(MPI_File *fh)
```

After I/O is performed by all processes, `MPI_File_close` is called collectively in order to close the file and return the filehandle.

The optimization potential of MPI derived datatypes and collective operations will be further illustrated in the following section by presenting Thakur et al.'s idea about four levels of realizing access patterns [TGL98] and giving an example of how patterns to access a distributed array can be optimized.

Four Levels of I/O Operations

Every application has a certain I/O access pattern. Nevertheless, depending on which I/O functions the application calls and how those are performed, this access pattern might be presented in different ways to the file system [TGL98]. Thakur et al. have classified I/O access patterns in four different levels: level 0 - level 3. Figure 5.3 explains this with an example of accessing a distributed array from a file. There is a two-dimensional array, which is distributed among 4 processes.

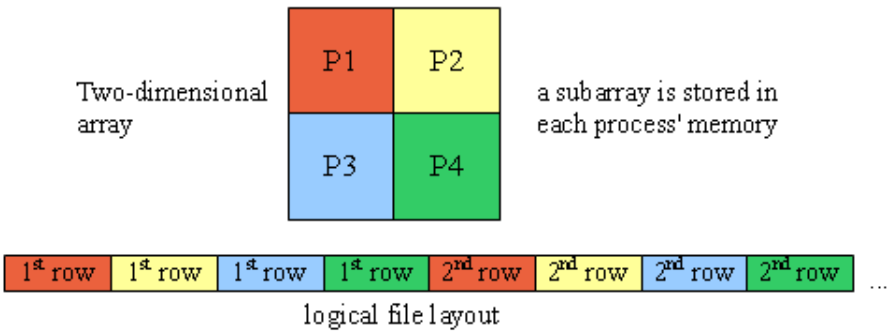

Figure 5.3.: Distributed array access

Each square in the figure stands for a subarray that is stored in the memory of the respective process. As it can be seen, the array is stored in the file in such manner that the subarrays of the processes are stored noncontiguously in the file.
Following this assumption, according to Thakur et al. [TGL98] there are four different ways of how this access pattern can be realized in MPI-I/O, corresponding to the four different levels mentioned before.

- Level 0 - independent read requests for every single row in the local array
  - many independent, contiguous requests (POSIX style)
- Level 1 - see Level 0, but using collective I/O functions instead of independent requests
  - many collective, contiguous requests
- Level 2 - each process creates a derived datatype to describe the noncontiguous access pattern, defines a file view, and calls independent-I/O functions
  - single independent, noncontiguous requests
- Level 3 - see Level 2, but using collective I/O functions
  - single collective, noncontiguous requests

5.2.1. Implementation of ROMIO

By using MPI's derived datatypes, ROMIO can perform any noncontiguous I/O request and not just on contiguous sections of arrays [TGL99a]. Additionally, for optimization purposes the implementation provides two parameters, by which users can define the number of processes that perform I/O and the maximum size of the temporary buffer for each process.

As mentioned before, most file systems use the POSIX API. In this section it will be presented how MPI-I/O is implemented in ROMIO and can increase performance even on file systems that use a POSIX API.

The basic POSIX I/O operations, like `open`, `close`, `read`, `write` and `seek` are also implemented in MPI-I/O. To open a file, there is the function `MPI_File_open`. In contrast to the POSIX `open` function, it is a collective function and is therefore performed by every process in the specified process group. In order to define such a process group, the user can use the MPI communicator argument. Since most file systems only support the POSIX `open` [TGL99b], ROMIO makes this function call for every process independently.

`MPI_File_close` closes a file and is identical to the `close` function in POSIX. However, there is a difference in behaviour if the file was opened with `MPI_MODE_DELETE_ON_CLOSE`, which results in deletion of the file after closing. Therefore, the implementation has to make sure that this happens in file systems that do not support this function, for example with the POSIX function `unlink`.

There are two `seek` functions in MPI-I/O, one for individual and one for shared file pointers. Most file systems do not support shared file pointers,

but do support the POSIX function `lseek`. ROMIO implements shared file pointers in such a way that it stores the value of the shared file pointer in a file. Every process then updates this value when accessing the file. This happens atomically and while locking the file.

Contiguous reads and writes can be implemented directly on top of the functions of the underlying file system. Noncontiguous file accesses, however, are more complicated to realize, since most file systems do not support noncontiguous file access. As mentioned above, in MPI, locations in files can be accessed noncontiguously by defining a file view and using derived datatypes. POSIX offers a possibility to perform several I/O requests at once with its `lio_listio` function. Essentially, this function provides a way to list multiple requests and then submit them in one function call, which makes it possible to access noncontiguous data in a file by submitting multiple requests to access the contiguous parts of the data. Nevertheless, every request will pose a separate nonblocking I/O request, so that an optimization for the whole list of requests is not possible in POSIX [TGL99b].
Therefore, in order to implement MPI's noncontiguous file access on file systems that do not support this, one could access every contiguous part of the requested data in separate function calls by using regular `read`/`write` functions. Nevertheless, as could be seen earlier, this will have a strong impact on performance, as it will result in many small, individual requests to the file system. In order to optimize performance, ROMIO uses data sieving instead to make noncontiguous access possible.

Similarly, if the file system does not support collective I/O, ROMIO uses Two-Phase collective I/O on top of the file system in order to improve performance. Studies by Thakur et al. have shown that data sieving performs only slightly better than Unix-style independent I/O on some machines while performing better on others, whereas collective I/O always had the best performance [TGL99b], which once again stresses the importance of collective I/O.

ROMIO supports two optimization techniques for I/O operations: data sieving and Two-Phase.

5.2.2. Data Sieving

Data sieving is a technique that uses MPI's ability of making noncontiguous file accesses in order to increase performance. If the user makes several small noncontiguous read requests to a file, the ROMIO implementation sums up these requests into one request in order to read a large contiguous part of the file instead. This large chunk of data is stored in a temporary buffer, where the needed data of the user requests is extracted and then copied into the user's buffer. This process is illustrated in Figure 5.4.

As can be seen, the user's requests for noncontiguous data from a file are merged into one function call that reads a contiguous chunk from that file, starting at the beginning byte of the first request and ending at the last byte of the last request (first and last in this case meaning according to the data layout of the file in memory). When this chunk is stored in the temporary buffer, the requested segments of the data are copied into the user's buffer. Even though with this technique more data is read than would be if several small requests were made, this is outweighed by gains in performance [TGL99a].

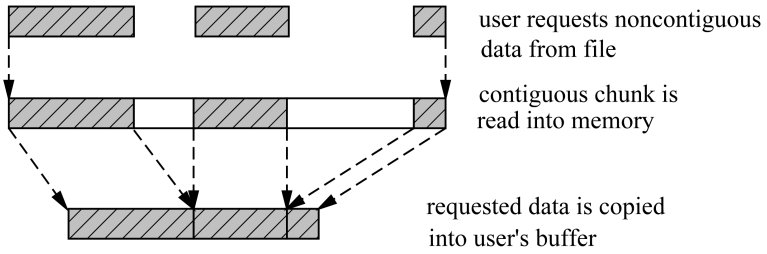

Figure 5.4.: Data sieving

There is, however, one problem to this technique. In order to store the requested data, a large temporary buffer is needed, as this must be able to store the whole extent of the user's request. This size can dramatically increase if there are holes in between the requested data segments [TGL99a]. ROMIO solves this problem by using a parameter by which the user can specify the maximum buffer size. This selected size corresponds with the size of contiguous data that can be requested by a process at one time. If the size of requested data is bigger than the maximum buffer size allows (the default value is 4 MB), data sieving is performed in parts [TGL99a]. For even more optimization potential, users can use hints in MPI-I/O in order to change the maximum buffer size during run time. However, it has to be kept in mind that if the holes are too large, any possible performance gains that could be achieved by data sieving are lost due to the cost of accessing the extra data.

Data sieving is not only possible for reading, but also for writing data. Different to reading, where no data is modified, it is important to perform a read-modify-write operation when writing data. This, plus locking the part of the file that is being accessed during read-modify-write, guarantees that no data is overwritten by other processes that access the file concurrently. Similar to the read operation, there is a parameter that allows users to specify the maximum buffer size for writing. Due to the lock that is necessary during writes, the default buffer size is smaller for writes (512 Kb) than the one for reads in order to reduce locking conflicts [TGL99a].

5.2.3. Two-Phase

As could be seen, performance can often be improved by merging mutliple small requests into one large request. In the case of data sieving, multiple requests are merged into a single independent MPI-I/O function to access data. Another, similar optimization technique is Two-Phase, which merges multiple requests into single collective functions to access data. ROMIO performs collective I/O at the client level and therefore uses a generalized version of the Two-Phase method [TGL99a].

To illustrate this Two-Phase method, we go back to our two-dimensional distributed array example (see Figure 5.3). As mentioned before, the data that has to be accessed by each process is located noncontiguously in the file. Consequently, when trying to access the needed data individually, every process will make several small noncontiguous read requests. Two-Phase tries to avoid this by using its knowledge about the entire I/O access pattern of all processes in order to make efficient function calls. In this case of the distributed array, it is important to see that all processes together read the entire file. Therefore, in order to achieve an efficient collective function call, the implementation has to have information about the entire I/O access pattern of all processes being involved.

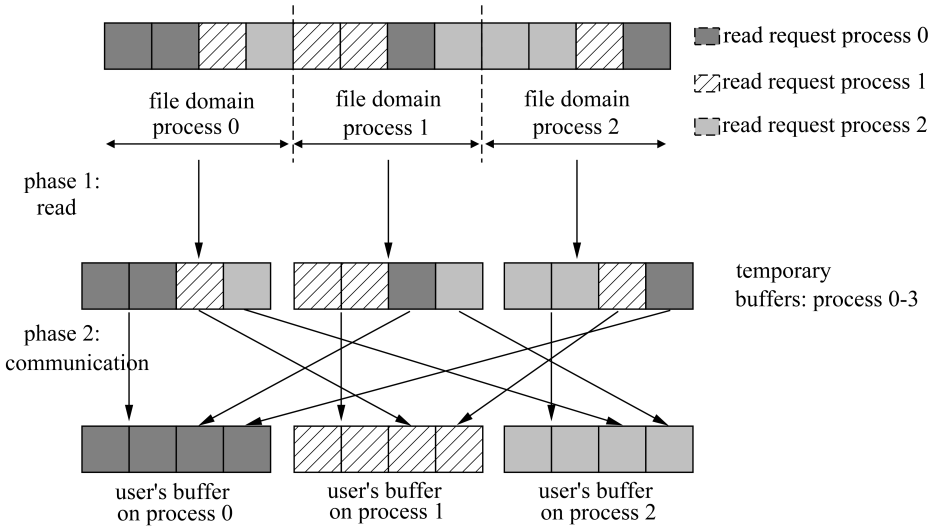

Figure 5.5.: A collective read with Two-Phase among three processes

As the name suggests, in Two-Phase file access happens in two phases: first, data file access and second, data redistribution among processes. At first glance it might seem inefficient to have a heavy communication phase after the actual I/O. This might be true, but the crucial point of this method lies in the first phase. Here, the file data is divided equally among the processes, so that every process will get an equally sized chunk of data. In the file access phase, every process makes a single, large, contiguous access, instead of a large number of small, noncontiguous requests, which reduces I/O time significantly. In the

second phase, interprocess communication happens, in which the processes redistribute the data in such a way that every process gets the requested data (as shown in figure 5.5). According to Thakur et al. this cost of interprocess communication is rather small when comparing it to the savings in I/O time [TGL99a].

Section 5.4 presents an I/O strategy, which further improves the performance of Two-Phase: *Layout Aware Collective I/O – LACIO*.

5.3. File System Support

The last sections illustrated how ROMIO implements MPI-I/O. It has also become clear that the underlying file system has a considerable impact on the performance of MPI-I/O. Based on these observations some features will be presented, which might be important to consider when designing a file system that could help to implement MPI-I/O with high performance, as suggested by Thakur et al. [TGL99b].

First of all, high performance access from multiple clients to a file must be supported. Not only do they have to be supported, but if there are concurrent requests for one file, the file system should be able to provide concurrent access to the file, while keeping the data consistent. Performance should not be hindered by strict locking mechanisms, which in some cases result in concurrent requests being serialized.
Going along with this, data consistency semantics have to be clearly defined, including the case how to handle concurrent accesses from multiple clients. Consistency should be guaranteed on byte-level or there should be a mode that supports byte-level consistency, meaning that modifications in a certain byte range must be visible to other processes immediately after return of the function, so that an explicit cache flush should not be required. These consistency semantics do not only have to be clearly defined, but also implemented correctly. This is for example provided if MPI-I/O is implemented with POSIX consistency semantics, as these support byte-level consistency.

Apart from data consistency semantics, the file system should also provide atomicity semantics and file attribute consistency. For better performance, atomicity semantics could be provided by allowing two modes: atomic and nonatomic modes, which can be chosen according to the application that is run on the file system. As most applications do not perform concurrent overlapping accesses [TGL99b], a nonatomic mode could be used, which increases performance. However, there has to be an atomic mode in order to provide consistency even for applications that perform concurrent overlapping accesses. In order to implement MPI-I/O's atomicity semantics, the file system should support a locking mechanism, like byte-range locking. ROMIO for example uses this in order to implement MPI-I/O's atomicity semantics for noncon-

tiguous file accesses, data sieving for write requests and shared file pointers [TGL99b].

File attribute consistency has to be supported in order to guarantee a consistent view on file attributes (e.g. file size) for all processes at any time. For example, if two processes open a new file and the first one writes some bytes to this file, if the second then calls a function to retrieve the file size, the actual file size should be returned. If attribute caching is allowed (as was the case in NFS [TGL99b]), it might happen that the second process receives the value of zero bytes as the file size, because outdated attributes (of the still empty file) are still in the cache. By not allowing attribute caching (for example with the "noac" option), this problem could be solved.
The file size attribute is especially important when the file system uses striping for storing data. As mentioned earlier, MPI-I/O allows the use of hints in order for the client to define certain file-striping parameters (such as the number of disks a file should be striped across and the size of the striping units). It might be important for a file system to provide "a good set of values" as the default, which allow to achieve good performance on a number of common access patterns and which can be modified by users on a per-file basis for even better performance [TGL99b].

The different access patterns of applications are the biggest issue that make it almost impossible to design an ideal file system which provides high performance I/O for all applications and implementations. Therefore, the file system should be able to detect and adapt to changing access patterns itself, or there must be an interface, which provides the user with a possibility to give more information to the file system about the application and access patterns being used [TGL99b].

The file system's interface should also be able to support noncontiguous file accesses. As stated earlier, MPI-I/O can perform noncontiguous I/O with the help of data sieving. However, this is done on implementation level, performance might be improved by letting the file system do this work [TGL99b]. Data sieving within the file system is done in a similar manner as caching: data is first written into the cache and afterwards the requested sections are retrieved by the file system, while the unused extra data remains in the cache and does not need to go to waste. If data sieving should be performed within the file system, it has to provide an interface that supports noncontiguous accesses. For this case, Thakur et al. have suggested an interface as the following (here for read requests, but similar to write requests) [TGL99b]:

```
int read_list(int mem_list_count, long long *mem_offsets,
    int *mem_lengths, int file_list_count, long long *
   file_offsets, int *file_lengths)
```

This interface simply consists of a list of offsets and lengths, `mem_offsets` and `mem_lengths` representing noncontiguous memory locations, while `file_offsets` and `file_lengths` represent noncontiguous locations in the file. The

`count` parameters contain the number of entries in the respective offsets and lengths buffers. In contrast to the POSIX `readv` and `writev` functions (also see 5.1), which can only be used to access noncontiguous parts in memory [LRT04], with `read_list` and `write_list` it is also possible to access noncontiguous regions in a file.

The next section gives an example of how a good collaboration between I/O interface and file system (in this case MPI-I/O and GPFS) might look like. GPFS uses its file partitioning mode in order to implement *data shipping* with MPI-I/O.

5.3.1. MPI-I/O on Top of GPFS

As an example to illustrate how well the interface and underlying file system can collaborate in order to achieve best performance, GPFS and MPI-I/O have been chosen. In GPFS, hints and directives for improving the file system's performance have been introduced. [PTH+01]. These hints and file system specific abilities, such as GPFS's data shipping and block prefetching, can be used by MPI-I/O/GPFS in order to make I/O more efficient.

Distributed locking is the default locking mode of GPFS. An alternative to this is file partitioning (data shipping) mode. When file partitioning mode is turned on, the file is partitioned in equal-sized chunks. Every chunk of the file is assigned to a certain I/O node, which alone is responsible for any I/O requests to this piece of data. I/O nodes only cache the file blocks that are assigned to this node, so data is never shared among the I/O nodes. Therefore, it is not necessary to use distributed locking, but rather to have one shared lock to a file, which is assigned to the responsible I/O nodes. This reduces locking conflicts and general locking overhead [PTB+00].

MPI-I/O/GPFS leverages this file partitioning mode for implementing data shipping. File partitioning and data shipping mode can work together in order to improve performance. The MPI-I/O/GPFS interface has knowledge about when it is best to use file partitioning. File partitioning is then performed by striping the file blocks on I/O agents in a round-robin fashion. A default stripe size is implemented, but can be modified by changing the parameters for file partitioning, so that file data can be mapped on I/O nodes as efficiently as possible [PTB+00].

For I/O requests there is now only one I/O agent, which is assigned to the file block, and will serve all requests regarding this file block. For reads, the I/O agent reads the data first and then ships it to the requesting process. For writes, the process issues its requests to the specified I/O agents that the file sections to be written are assigned to. These I/O agents are then responsible for performing the writes on their assigned file blocks.

MPI-I/O and GPFS data shipping can work closely together in order to improve performance. With MPI-I/O the data can be shipped most efficiently from the requesting nodes to the GPFS I/O nodes and the other way around, while GPFS can improve performance by being able to make larger I/O requests (as file blocks are mapped efficiently) and use shared locks [PTH+01].

5.4. LACIO

Two-Phase achieves I/O optimization by splitting I/O up in two parts: the actual I/O and communication. While this increases performance by splitting up and redistributing data, thus avoiding small, noncontiguous requests, this technique could be further improved if not only the logical layout of the file would be taken into account, but also the physical layout of the file at disk level. Chen et al. have developed a new I/O strategy called *"Layout-Aware Collective I/O"* (LACIO) [CST+11]. This strategy is based on Two-Phase, enhancing it by adding the important factor of the physical data layout.

As can be seen in Figure 5.6, the logical layout of a file usually does not match the physical layout, as files are striped across disks. This allows potential for further optimization. Figure 5.6 shows a file, which is striped across three servers in a round-robin fashion. Striping is used in order to improve I/O performance in a parallel environment. In usual collective I/O that uses Two-Phase, this file would be split up into three parts (as marked by the dotted lines in Figure 5.6) in order to distribute the chunks equally among the three processes (it is possible to change the number and size of the chunks by modifying the parameters that are offered by the MPI-I/O interface). Consequently, every process is assigned a file domain. Usually this is done by the processes themselves in a first round of communication by calculating the total length of file access that is needed and then dividing this among all processes. In this example, I/O would then be performed as shown by the dashed arrows. Every process would have to retrieve data from two storage servers, store this data in its temporary buffer (supposing the size of the buffer is sufficient), and, depending if this is necessary, exchange data with other processes.

Depending on how well matched the logical and physical layout of the file is, this proceeding might lead to increased access contention [CST+11]. If only the logical layout of the file is taken into consideration, potential gains in performance might be lost due to resulting concurrent requests to all file servers [CST+11]. Therefore, the physical distribution of a file across the storage devices should be taken into account, as this might further improve performance significantly. What Chen et al. propose is a new strategy, which in addition to usual Two-Phase, rearranges the partitions of file domains and the processes' requests before the I/O phase, so that requests are: *grouped*, so that they need as few file servers as possible, and *reordered* to not only be logically,

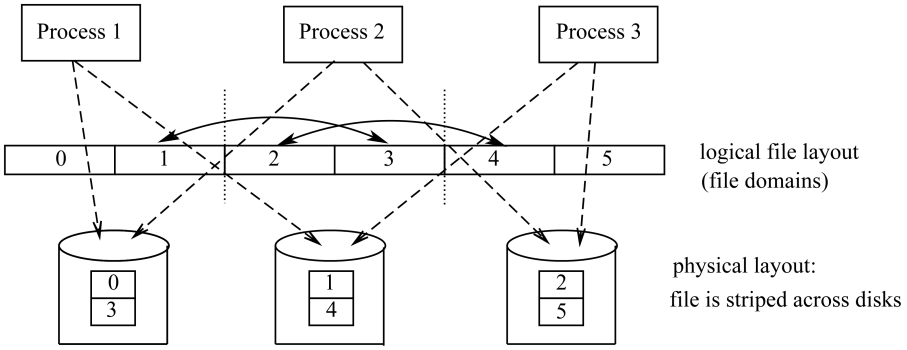

Figure 5.6.: LACIO can improve I/O by reordering the logical layout of the file to better match the physical layout of the file.

but also physically contiguous [CST+11]. This promises to achieve better concurrency, reduce access contention and exploit better locality [CST+11]. However, in order for this strategy to work, it has to be possible for the I/O implementation to receive information about the physical data layout of the underlying file system through some kind of interface. PVFS for example already provides such an interface [CST+11].

In the example in figure 5.6, the rearrangement is done by reordering the requests for some (or every) process(es). In this phase, data is not yet exchanged between processes. After this phase of reordering, the actual I/O will be performed. As illustrated in figure 5.7, the rearrangement of the file domain's partition and requests has helped to decrease the number of I/O accesses. Without LACIO, six accesses would have been necessary, with every process performing I/O with two storage servers. With LACIO, it was possible to reduce the number of I/O calls to three, with every process communicating with only one storage server.

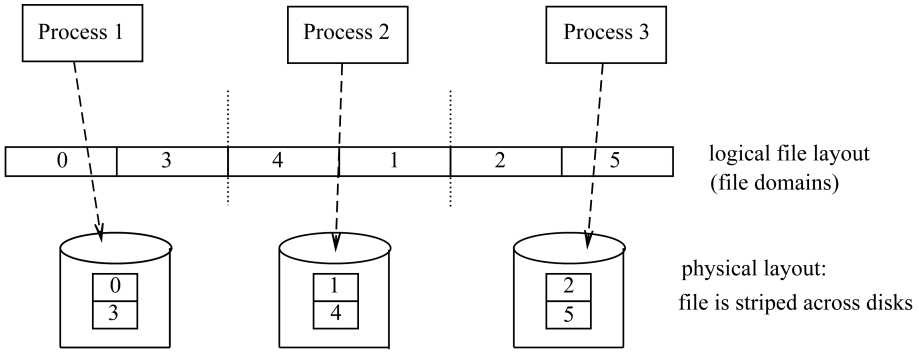

Figure 5.7.: Reordered requests match with the physical layout.

It has been apparent that the LACIO strategy can improve I/O performance quite significantly, however, it has to be examined how much overhead is produced by the rearrangement process and if this is worth the effort at all. It is not only the rearrangement process, which produces extra overhead, but also retrieving the information about the physical file layout from the file system

API, which has to be taken into account. Chen et al. have found that the best solution for communicating the data layout information is by receiving the information once from the file system API and then broadcast it to every process, rather than letting every single process do this itself. This is done for the first time when a collective I/O is first performed, and the layout information is then stored in the cache of every process (this is safe as the information is static and will only change if modified explicitly or deleted) [CST+11].

Summing up, LACIO provides a rather new approach to I/O optimization. However, it still has to be shown if it achieves better performance even if logical and physical file layout do not match well at all. In this case reordering overhead might be quite severe, in addition to communication overhead in order to retrieve information about the physical file layout in the first place. Another critical point is the fact, that I/O access patterns may change a lot even in the same application. Therefore, in order for this I/O optimization technique to work, these changes have to be taken into account, as reordering has to happen every single time.

5.5. An Ideal Interface for HPC

The defining factor for applications in HPC is that huge datasets have to be processed, while providing the best performance possible. While the calculation part is usually not a problem for supercomputers these days, as the speed of processors is constantly increasing, it is the I/O, which poses a bottleneck in the system and hinders performance. The I/O interface which serves as a connecting layer between application and underlying file system plays a big role in this, as the overall performance of the application can only be as well as the interface allows.

As mentioned before, POSIX I/O is still the standard interface to most file systems, even though it is not the ideal choice. The list of extensions to the POSIX interface promises improvement, for example by adding more functions, however those are just, as they are called, *extensions* to the crucial problem. Those new functions are based on the original ones, which are hardly compatible with HPC anymore. The idea comes to mind that it might be better to build a new interface from scratch, and implement this as a new standard, instead of extending an old one.

In order to make suggestions about how a new interface might look like, it is important to find out, how I/O in HPC usually works, and if there is a recurring pattern. Studies have shown, for example by Thakur et al. [TGL99a], that a typical pattern can be made out in HPC applications: they work with huge datasets, many accesses on the same file, while only few overlapping accesses in the same file. As overlapping file access is not used the most in HPC, the ideal interface should be able to handle collective I/O operations, so that

non-overlapping write requests and overlapping read requests to the same file can be passed on to the file system in one request. This is another reason why POSIX is not the best choice, as it does not provide proper collective functions.
Going along with this, there should be a way to handle proper noncontiguous file access, as it is possible with MPI-I/O. As it could be seen earlier (compare section 5.1) POSIX is behind MPI-I/O in this respect. POSIX provides `listio` in order to acces noncontiguous parts in a file. However, as the name suggests, this is just a list of requests that is passed on to the file system in one request, so that the file system still has to handle all of the requests individually. This might reduce the number of function calls to the system, but does not take a lot of work off the file system. Noncontiguous file access is better handled in MPI-I/O, which provides the possibility to create datatypes and pass these datatypes on to the file system, so that real noncontiguity can be achieved. Of course, the ideal interface between application and file system has to be able to support the use of such datatypes.

Another point to consider is metadata handling. The question that comes to mind is, how important is metadata actually? As has been pointed out a couple of times in this thesis, some metadata is very important in order to maintain data consistency, like for example the last modification time, `mtime`, but also the exact file size, which is important for read requests over the end of a file. However, it is not useful, as it is for example handled in POSIX, to give out all metadata about a file with every `stat` call. Nor is it useful to update the file's metadata every time a read to the file happens (`atime` is updated every time a file is accessed), as this results into many calls to change metadata to the inode. This is especially complicated, if metadata is managed centralized, as it is the case in GPFS. For improvement, POSIX has suggested a modified `stat` call, `statlite`, in order to retrieve lazy metadata attributes. In contrast to `stat`, `statlite` does not fill some attributes at the same granularity, like the file size for example, or not at all if not specifically requested, such as `atime` and `mtime`. Studies have shown that the use of `statlite` increases performance [VLR+08], however, there is still optimization potential. For example, it is not possible to determine exactly what attributes to return when requesting metadata, for example by providing a list of the wanted attributes as arguments to the call. This might be a task for the ideal interface. It should be able to retrieve the needed information with one single request.

As it has become clear in this thesis, there are different I/O optimization techniques, which have been developed for different underlying file systems, like for example data shipping for MPI-I/O on top of GPFS. Just in the same way as there have been suggestions in order to improve the existing I/O interfaces, like the POSIX extension goals. All of this goes to show that there is no consensus about what is best and should be the standard in HPC. However, the majority of opinions seems to go into the direction of MPI-I/O. MPI-I/O could be seen as the de-facto standard in HPC, however this does not solve the problem that the standard of the underlying file systems is still POSIX.

Taking all of this into account, ADIOS seems to be a good first approach in order to tackle this problem. What ADIOS provides is a simple way for the client to choose the fitting I/O interface to use for every individual application. Therefore, ADIOS provides portability, which is a good start in the process of finding the ideal interface, as it makes code portability between different interfaces possible. However, ADIOS also adds another layer on top of the I/O process, which should be avoided. Nevertheless, until the perfect I/O interface has been designed, ADIOS is a good alternative.

Summary

The focus of this chapter has been on I/O optimization techniques. An overview has been given over possibilities to improve POSIX I/O and MPI-I/O, while showing how ROMIO implements MPI-I/O. Before making suggestions about how the underlying file system could further improve performance, an example was given by showing how GPFS's data shipping works. In addition to this, LACIO has been presented, a technique that takes into consideration the physical data layout of a file. This last section has given a good overview of how an ideal I/O interface for HPC might look like. Based on these thoughts, ADIOS seems to be a step into the right direction and provides a useful alternative until the ideal interface has been designed. The next chapter will further encourage this impression. Based on the information that has been gathered in this and the previous chapters, the performance of ADIOS together with MPI-I/O and POSIX I/O in combination with two file systems will be examined in a real application.

Chapter 6

Evaluation

The previous chapters have given information about what factors play a role in I/O performance, as well as how certain I/O techniques may further improve performance. This chapter will be an evaluation of the previous thoughts. For this reason, the performance of three I/O interfaces, POSIX I/O, MPI-I/O and ADIOS, in combination with two file systems, NFS and the parallel file system PVFS, has been examined. An overview of the experiments will be given, concluding with an interpretation of the results.

6.1. The Experiments

In order to examine I/O performance, the program `partdiff-par` was used. `partdiff-par` is a program, which can be used in order to solve the Poisson equation with an iterative algorithm. Not much detail will be given about the mathematic realisation here, as the focus primarily lies on the I/O pattern of this program. In order to compare the performance of different I/O techniques, the program has been implemented for three I/O interfaces: MPI-I/O, POSIX I/O and ADIOS.

The solution of partial differential equations plays a big role in many (high performance) scientific applications. Some examples include the conduction of heat or the propagation of sound, which can all be generally described with the help of partial differential equations. Therefore, finding ways for solving these complex equations is important to research.
The Poisson equation is one kind of partial differential equation, which is often used in mathematical engineering and physics. One possible way for solving such an equation is by using an iterative algorithm. The `partdiff-par` program provides such an iterative algorithm. It can be used to solve the Poisson

equation for two disturbance functions. In order to be able to focus on the topic of interest, I/O performance, rather than the calculation process, the algorithm presented here was used with the disturbance function f(x,y)=0. Calculations are done in a matrix, which is stored row-like in memory. The application was parallelized in such a way that the data of the matrix is distributed equally among all processes, every process receiving a number of rows of the matrix, which are stored contiguously in memory.

I/O does not play a big role during the calculation process. It does come into play, however, when data has to be written to disk. This happens at different points during execution. Once, of course, at the end of the program run, when the result (the matrix) has to be stored in a file. Just as important is backing up intermediate results from time to time, as it is always possible (especially in high performance applications that work with huge datasets and long runtimes), that the system fails during runtime. For this case, it is important to have a backup of data from some point prior to the failure, so that it is not necessary to roll back the whole application and restart from the beginning, which would mean a big workload depending on the size of the data that was to be processed. The `partdiff-par` program writes checkpoints to disk at specified intervals, so that in the case of a failure, the program can be taken up again from the last checkpoint. At program start, it can be specified in a parameter at what intervals a checkpoint should be written to disk. There is always one checkpoint available (the one that has been written last). When it is time for a new checkpoint, the one prior to the last checkpoint will be overwritten, so that there is one complete checkpoint available at all times, even if a failure happens during checkpointing.

As can be concluded, the I/O pattern is the following: all processes write their section of the matrix to disk. This means a lot of concurrent write requests to the same file (the one that will store the checkpoint), but no overlapping write requests, because the rows of the matrix are split up equally among all processes, so that no row belongs to more than one process. The chunks of data that every process has to write are contiguous, as the matrix is stored row-like. This would be different, if the data was stored column-like on the disks. Therefore, there are many concurrent, non-overlapping, contiguous write requests from every process.

In order to examine the performance of I/O interfaces, the program has been implemented for three different I/O interfaces. The original program was using MPI-I/O. Listing 6.1 shows the part of the program that deals with I/O – the `writeCheckpoint` function, which writes a checkpoint of the current matrix to disk.

Listing 6.1: writeCheckpoint function using MPI-I/O

```
1  static void writeCheckpoint (void)
2  { //...
3  //open file
4  MPI_File_open (MPI_COMM_WORLD, name, MPI_MODE_WRONLY |
       MPI_MODE_CREATE, MPI_INFO_NULL, &fh);
5
6  #ifdef COLLECTIVE
7    ret = MPI_File_write_at_all ( //collective write
         function
8  #else
9    ret = MPI_File_write_at ( //noncollective write
10 #endif
11   fh, //MPI_File fh
12   startPos, //MPI_Offset offset
13   Matrix[m1][pos] + curElement, //void *buf
14   writeElements, //int count
15   MPI_DOUBLE, //MPI_Datatype datatype
16   &status //MPI_Status *status
17     );
18 MPI_File_close (&fh);
19 }
```

Upon the file opening call (`MPI_File_open`), a file handle (`fh`) is retrieved, which can be used in order to perform I/O to the file. If the file does not yet exist, it will be created when the `open` function is called for the first time. When using MPI-I/O the user has the choice between a collective and a noncollective `write` function. Presumably, the collective call will achieve higher performance than the noncollective call, as in this case only one request for all processes is made to the file system. This will be examined later. `MPI_File_open` and `MPI_File_close` are both collective functions, which have to be called by all processes in the communicator.
Another advantage to POSIX-I/O is the use of datatypes. Here, the `MPI_Double` datatype is used in order to write an 8-byte double, which is represented as one element to the file system, whereas in POSIX-I/O a double is a block consisting of eight bytes. How much the performance gains are with MPI-I/O compared to POSIX-I/O, will become clear in the next section when the results are presented.

Listing 6.2: writeCheckpoint function using POSIX I/O

```
1  static void writeCheckpoint (void)
2  {//...
3  //open file
4  fd = open(name, O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR|S_IRGRP
       |S_IROTH);
5
6  //write data to file
7   pwrite(
8              fd, //int fd
9              Matrix[m1][pos], //const void *buf
10             mycount, //size_t count
11             startPos //off_t offset
12             );
13
14 close(fd);
15 }
```

Listing 6.2 shows the same I/O routine for POSIX-I/O instead of MPI-I/O. `pwrite` does not have an equivalent collective function, so that it has to be called by every process individually. Similar to MPI-I/O, the `open` call retrieves a file handle either to the file if it exists, or it creates a new file and returns its file handle. At the same time, this new file's access rights have to be issued with the last parameter.

It will also be interesting to see, whether the implementation with ADIOS will further improve performance. Listing 6.3 shows how I/O is implemented with ADIOS. Before ADIOS can perform operations, it has to be initialized, which usually happens towards the beginning of the main method. The XML file will be parsed, the specified memory allocated and the transport methods defined. The XML file in this case is the same as the one provided earlier in section 4.3, and is therefore not listed again here.

ADIOS opens the group, which is associated with the file handle, by the `adios_open` call, upon which a filehandle is returned (`fd_p`). The group name is defined in the XML file. `name` is a simple string, which represents the file name. The file access mode can be assigned to the file with parameters "w" (write or create if it does not exist), "r" (read) or "a" (append file). If no coordination among processes is needed (by the transport method specified in the XML file, as it is for example the case with POSIX, in contrast to MPI), ADIOS opens the file, otherwise it will be opened in the specified communicator.

`adios_group_size` defines the group size and passes it to the internal ADIOS transport structure. Eventually, `adios_write` performs the write operation, writing the elements from the buffer issued as last argument to the buffer defined by the variable. This buffer size can be specified in the XML file. If it is big enough, ADIOS only copies the data to the buffer and writes to disk

during calculation, otherwise it will write to disk directly.

Listing 6.3: writeCheckpoint function using ADIOS

```
1  static void writeCheckpoint (void)
2  {//...
3  adios_open(&fd_p, "checkpoint", name, "w", &comm); //open
       group, filehandle fd_p,
4    // group_name "checkpoint", file_name name, file access
        mode "w" (write),
5    // and communicator comm
6
7  adios_group_size(fd_p, (myrows*(N+1)*8), &total_size); //
     define group size,
8    // myrows = number of rows that are assigned to the
        process
9    // (N+1) = number of columns in the matrix (N=
        interlines)
10   // total_size is the total group size
11
12 adios_write(fd_p, "matrix", Matrix[m1][pos]); //ADIOS
     write,
13   // "matrix" is the variable defined in the xml
14   // last argument points to a buffer which stores the
        elements to be written
15
16 adios_close(fd_p); //close file
17 }
```

In addition to the performances of the three I/O interfaces, it was also interesting to see if and how two different file systems achieve different performances. For this purpose, the (nonparallel) network file system NFS was used as underlying file system in the first part of the experiments, while the parallel file system PVFS served as file system in the second part.

The cluster, which was used for the test cases, consisted of one master and ten compute nodes. Each node has two processors with six cores each and two 1 gigabit Ethernet networks (of which only one is used at a time). NFS was served from the master node and mounted on all ten compute nodes, while five compute nodes served PVFS. The underlying disk has a highest possible throughput rate of approximately 115 MB/s for read access and 100 MB/s for write access. However, these are ideal values, which will most possibly not be achieved in reality.
The application was run on a number of compute nodes (n) while scaling the number of processes per node (ppn) and using a variety of I/O interfaces. An overview of the results will be given in the next section.

6.2. The Results

The application was run on the cluster with the following parameters: 200 interlines[1] and 500 iterations as termination condition. In order to get a good overview about how I/O influences performance in the test cases, a checkpoint of the size of 20.71 MB is written to disk every second iteration. After 500 iterations and therefore 250 `writeCheckpoint` function calls this adds up to a total data size of 5177.5 MB, which is written to disk in every case. The checkpoint size of 20.71 MB per iteration can be calculated with the following equation:

$$checkpoint\ size = (number\ of\ rows) * (number\ of\ colums) * (size\ of\ double)$$

which in this case translates to

$$checkpoint\ size = (200*8+9)*(200*8+9)*8 = 20,711,048\,\text{Bytes} \approx 20.71\,\text{MB}$$

The amount of data that is written by every process is divided equally among all processes. For example, if there is only one process, it will write the whole 20.71 MB checkpoint to disk every second iteration, while if there are two processes (on the same or on different nodes), the data chunk is divided among them, so that every process only has to write approx. 10.36 MB every other iteration.

There are a number of factors that can influence the speed of I/O in a cluster, like other applications running at the same time, background activity, or fluctuating influences from the operating system or the application itself. Therefore, in order to keep possible influences from concurrent applications at a minimum, the cluster was blocked for other applications during the time that the experiments were running. Despite of this precautionary measure, it was already apparent after the first few test runs that the runtimes would still fluctuate considerably, even if the application was started under equal conditions (the same number of nodes and processes and identical parameters). For this reason, every test case was repeated several times. The results shown in the tables are the arithmetic averages of all runs under comparable conditions. An overview of all test cases including the total application runtimes (performing calculation + I/O) is presented in the appendix, tables A.1 and A.2.

For NFS there are 16 different test cases: 12 cases running on one node, while scaling the number of processes starting with 1 up to 12. And two test cases each with 5 nodes and 10 nodes (for 1 and 12 processes). The interfaces that have been examined are first, MPI-I/O, collective and non-collective, second, POSIX-I/O, once writing data directly to disk on the underlying file system (NFS) and once writing data to local storage on the same node (tmp),

[1] the matrix size is calculated like this: size=interlines*8+9, whereas (N+1) = number of columns/rows in the matrix, with N = interlines*8+8

and third, ADIOS with different transport methods: MPI, POSIX and NULL (which performs no I/O). In order to be able to directly compare the I/O performances of all three interfaces, the runtimes presented in table 6.1 show the times that were exclusively used for performing I/O, whereas table 6.2 illustrates the comparison between application runtime without performing I/O and with performing I/O to local storage. Total application runtimes, including the time spent calculating, can be found in the appendix (tables A.1 and A.2).

n	**ppn**	**POSIX I/O**	**MPI-I/O**		**ADIOS**	
			non-coll.	coll.	POSIX	MPI
1	1	235,86	228,65	245,75	25,60	35,42
1	2	297,21	242,24	262,38	76,37	48,15
1	3	227,18	226,22	255,78	64,01	19,71
1	4	265,33	272,09	268,33	17,38	51,38
1	5	252,25	244,85	271,26	77,36	35,56
1	6	275,94	271,36	271,4	42,87	83,69
1	7	258,51	246,04	283,74	105,34	45,74
1	8	271,91	275,71	287,75	15,53	67,59
1	9	283,85	290,12	303,55	53,90	88,84
1	10	245,38	263,98	258,25	58,11	83,61
1	11	280,78	282,51	269,66	39,52	74,88
1	12	273,52	266,47	310,58	51,18	63,44
5	1	261,73	282,98	277,91	60,46	71,17
5	12	607,60	604,7	655,24	265,88	73,62
10	1	273,69	380,71	321,24	33,65	68,53
10	12	851,17	813,89	851,19	456,54	354,99

Table 6.1.: I/O time in seconds, writing 5177.5MB using NFS.

The results in table 6.1 show that there are no striking differences when comparing the performance of POSIX and MPI-I/O (collective as well as non-collective) in combination with NFS. In fact, none of the three performed better in all cases than the other two. There also does not seem to be a pattern showing which one achieves better performance with an increasing number of processes. What is quite obvious is, that I/O takes longer the more processes are participating, which is understandable when keeping in mind that all processes have to access the same file concurrently. The network poses a bottleneck, and so does NFS, which has to be able to synchronize all file accesses. Its lack of parallelism plays a significant role here. With an increasing number of processes the data throughput decreases immensely (see figures 6.1 and 6.2 at the end of this chapter for further illustration), even though the amount of data that is being written to disk stays the same in all cases. This is even more apparent if the same number of processes runs on one node or on several nodes with only one process per node. For MPI-I/O and POSIX I/O in these experiments it is always the case that a number

of accesses coming from one node takes less time to perform than the same amount of accesses coming from several nodes.

When comparing MPI noncollective and collective I/O, it is surprising that noncollective I/O often performs better than collective I/O. However, this might be explained by the application's file access pattern, which is accessing rather big contiguous file parts. On the one hand, the more processes are involved here, the less data per process is to be written to disk, which overall increases performance. On the other hand, if these processes perform I/O collectively, while accessing large contiguous file parts, noncollective I/O might have better performance, because of the independence of the write calls. When writing collectively, it might be possible that some processes have to wait for others, which are still in the calculation process, which slows down the application. A solution for this might be implementing I/O asynchronously.

The most striking observation in these experiments, however, is the extraordinary performance that is achieved by ADIOS. Independent from the chosen transport method, ADIOS achieves best performance in all test cases, sometimes only needing a tenth of the time of the corresponding interface. One possible explanation for this obsvervation might be the internal buffer that ADIOS uses for storing data. When there is enough buffer space available, ADIOS does not write to disk immediately, but into the buffer, as long as there is space available. Decreasing the buffer size might result into decreasing the speed of I/O.
Comparing the different transport methods used, there is no regular pattern recognizable. Oftentimes MPI performs even worse than POSIX, which is astonishing. Furthermore, there does not seem to be a pattern how performance scales with an increasing number of processes, although it seems that ADIOS with MPI is slower the more processes run (as long as the application runs on a single node). What seems to be the case for both, MPI and POSIX as transport methods, that an increasing number of nodes increases performance, while an increased number of processes on the same number of nodes slows down performance. This is contrary to the observations made in the cases that ADIOS is not used.

Table 6.2 presents an overview of the application's runtime when no data is written to the file system. Column three shows how the application performs with the I/O function commented out, while column four shows the results for ADIOS using transport method NULL (meaning no I/O is performed). These results are compared to the application's runtime when performing I/O to a local storage device (on the same node as the processes). This comparison goes to show how quick local storage access is, namely only a few seconds whereas I/O to a distributed storage device is restricted by the speed of the network and file system.

n	ppn	without I/O	ADIOS NULL	POSIX tmp with I/O
1	1	82,86	83,13	82,83
1	2	49,48	52,05	53,55
1	3	38,26	39,54	37,67
1	4	24,92	26,95	26,49
1	5	19,39	25,00	21,51
1	6	18,58	18,91	17,39
1	7	18,70	16,72	16,57
1	8	15,59	14,34	14,14
1	9	14,37	14,92	12,92
1	10	15,09	20,31	12,07
1	11	11,82	11,96	11,46
1	12	10,98	12,08	10,42
5	1	19,42	19,18	–
5	12	5,52	3,16	–
10	1	9,85	9,9	–
10	12	3,25	2,67	–

Table 6.2.: Application runtime without I/O, ADIOS NULL and POSIX I/O using local storage: 200 interlines and 500 iterations

When comparing ADIOS using 'NULL' for transport method to POSIX using local storage on the same node that the processes are running on, it becomes clear how little time the I/O part takes. Of course it has to be kept in mind that there might be other factors on the cluster (as the ones mentioned before), which can influence the overall performance of the application, because otherwise the column with the 'NULL' values could never be higher than the values in the 'tmp' column of the same row. This goes to show that I/O to local storage takes up only very little time, especially when comparing it to the times used to write data to the underlying file system instead.

For better comparison, another test run was done, running the application with MPI-I/O but manually having commented out the I/O function calls (see 6.2, 'without I/O'). This goes to show that the results of running ADIOS with 'NULL' method and MPI without I/O operations, which should be the same in theory, actually differ in reality, which once again stresses how much of an influence other factors such as background activity on the cluster have on the application and consequently the test results presented here.

In the second part of the experiments, the underlying file system was PVFS instead of NFS in combination with two interfaces: MPI-I/O (collective and non-collective) and ADIOS (transport method: MPI). The results can be seen in table 6.3.

n	**ppn**	**MPI-I/O**		**ADIOS**
		non-coll.	coll.	MPI
1	1	59,88	59,39	38,09
1	2	71,09	70,47	48,62
1	3	62,17	63,07	48,66
1	4	66,50	66,79	47,70
1	5	67,51	67,19	46,70
1	6	66,28	66,22	43,04
1	7	69,53	70,43	37,98
1	8	74,84	74,74	41,17
1	9	77,25	77,38	37,99
1	10	79,35	79,93	38,35
1	11	84,22	84,53	41,46
1	12	87,25	89,44	42,26
5	1	53,88	57,17	36,98
5	12	83,21	80,77	37,78

Table 6.3.: I/O time in seconds, writing 5177.5MB using PVFS.

What is striking at first glance is the little amount of time that is spent with I/O when PVFS, compared to NFS, is used. Due to PVFS's parallelity I/O time can be decreased by an approximate factor of at least four (for MPI non-collective and collective). The performance of ADIOS, however, does not increase by an equal factor. In fact, there is not much difference in performance between NFS and PVFS in combination with ADIOS as long as the application runs with few processes on one node. With an increasing amount of processes, however, there are gains in performance noticeable. Especially from seven processes onwards, the time spent on I/O is decreasing with PVFS, reaching performance gains of approximately 1.5 to almost 2 compared to NFS. Unfortunately, for the test cases PVFS was only available on five compute nodes, posing a limit of 60 processes in total. If run on even more nodes with more processes, performance with ADIOS could presumably be improved by an even higher factor.

Furthermore, while with NFS I/O times for all interfaces seemed to fluctuate, with PVFS there is a pattern emerging. The more processes are involved, the more time is spent with I/O, when using MPI-I/O. ADIOS, however, is able to fully use the PVFS's advantage of parallelity, so that I/O time maintains relatively constant even with an increasing number of processes.

Figure 6.1 provides a good illustration of the throughput that can be achieved by both file systems. The number of processes shown in figure 6.1 are all running on one node. It goes to show that ADIOS with MPI (red and green bar) achieves almost always highest throughput in PVFS as well as NFS. As mentioned earlier, ADIOS even performs a bit better on NFS when few processes are involved. Nevertheless, the more processes are participating, the more its throughput seems to decrease. The throughput that ADIOS can achieve

in combination with PVFS, however, stays relatively constant (compared to NFS), no matter how many processes are involved. The arithmetic mean of the achieved throughput by ADIOS with NFS and PVFS is illustrated by the two dotted lines in figure 6.1, which makes it clear that ADIOS in combination with PVFS achieves the highest throughput on average.

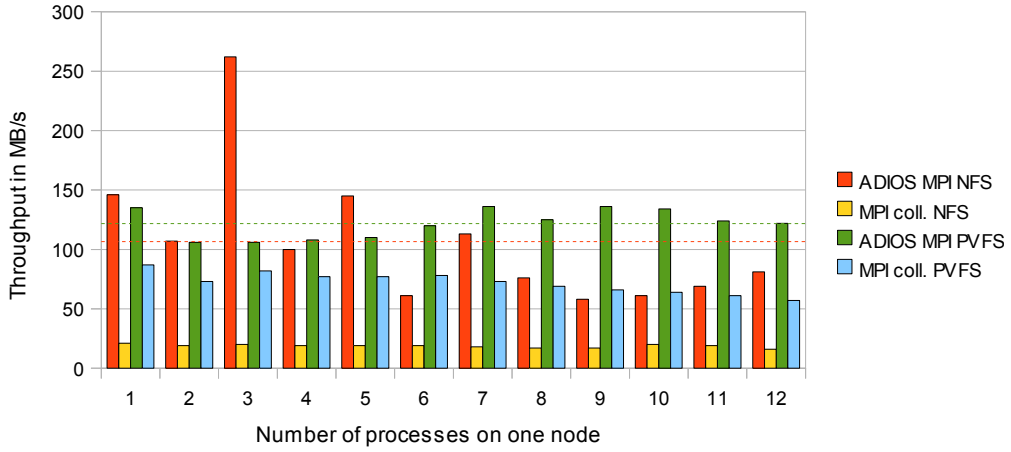

Figure 6.1.: Comparison of throughput: NFS and PVFS. All processes running on one node

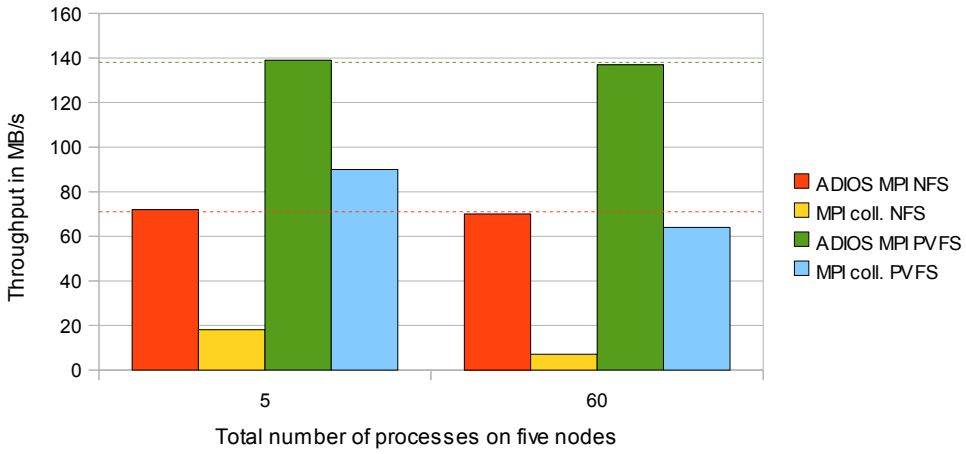

Figure 6.2.: Comparison of throughput: NFS and PVFS. Processes running on five nodes.

Figure 6.2 shows a similar graphic, this time the processes were running on five nodes, with one and twelve processes each. This goes to show that ADIOS in combination with PVFS reaches constantly high throughput even with a total of 60 processes, 12 processes on five nodes each. That more nodes are involved does not have an influence on the throughput for ADIOS and PVFS, while the throughput achieved with ADIOS and NFS is stable, but on average less than the one achieved when all processes are running on a single node. As

expected, throughput of MPI-I/O falls off with an increasing number of nodes, on NFS by roughly 50%, whereas less than 20% on PVFS, comparing throughput achieved with one process on five nodes each to throughput achieved with twelve processes on five nodes each.

Overall, it can be seen that simple MPI-I/O collective together with NFS achieves very poor throughput, while the same routine with PVFS can reach acceptable throughput, which sometimes even exceeds the throughput that has been achieved by ADIOS on NFS with the same number of processes. However, the performance and throughput achieved by ADIOS (on both file systems) can not be beaten.

Summary

These experiments have shown that both, I/O interface and file system, play an important role in what performance can be achieved by an application. In order to fully utilize an application's potential for performance, the file access pattern has to be kept in mind. Therefore, as could be observed, collective MPI-I/O does not always achieve better performance than non-collective if the access pattern does not take advantage of collective operations. ADIOS clearly performed best in all cases, however the influence of the buffer size has to be kept in mind.

In addition to this, it has also become clear in the second part of the experiments that an I/O interface can not achieve ideal performance if the underlying file system does not provide any support. This was the case with MPI-I/O, which achieved remarkable better throughput with PVFS instead of NFS.

Chapter 7

Conclusion and Future Work

In this thesis, the wide area of I/O optimization in high performance computing has been tackled. By taking a look at most of the factors that influence I/O performance – file systems, semantics and interfaces – an overview has been provided of what can be done in order to aim for best performance.

As it has been shown, the file system design plays an important role, as this is the resource that provides an application with the data it needs. Depending on its implementation and design, it can either pose a bottleneck, or support the application. Just as much an impact the file access semantics have. The file system has to be able to deal with many concurrent requests, while the semantics provide the necessary information about how these should be handled. Stricter access semantics might pose a threat to performance, while relaxed semantics might result in inconsistent data. Therefore, it is important to find the right measure when implementing these semantics.

This is where the I/O interface comes into play. It provides sets of I/O functions for implementation. As has become clear, collective I/O operations, such as provided by MPI-I/O, can achieve good performance, especially when many processes access non-overlapping sections in the same file. Some I/O techniques can further optimize performance, such as data sieving and Two-Phase.
However, at the same time, it has to be made sure that the overhead, which might be produced by some of these techniques, are worth the increase in performance. As an example, Two-Phase does not always perform well, especially if the physical layout of the file on the disks does not match the logical one very well, which is rather often the case. LACIO does take such factors into account when aiming for better performance, nevertheless how well I/O with LACIO performs is strongly influenced by how much overhead is produced when retrieving the physical file layout from the disks in order to take reorder-

ing measurements. Additionally, it has to be taken into account that access patterns might change quickly between different applications, which also means that the optimization techniques has to be modified just as quick in order to keep performance on a consistent level.

The evaluation part of this thesis has shown that the I/O interface and file system have a great impact on an application's performance. PVFS achieves overall better performance than NFS due to its parallelity, while ADIOS clearly reaches best performance in almost any case. One reason for ADIOS's great performance is the possibility to define a buffer size, which allows simply copying data to this local buffer during I/O and writing to disk during calculation. This also goes to show the actual advantage of ADIOS: keeping I/O separate from the rest of the application's source code. This makes it possible for the user to easily modify the I/O configurations and make them fit to the individual application, even allowing different I/O methods to be used for different I/O operations within the same application.

Concluding, one can say that there is no perfect solution yet, which guarantees to achieve the best performance at all times. It is always an interaction between all partaking resources. Even though there are optimization techniques, those have to be used efficiently and with consideration to the application's dominant I/O patterns as well as the underlying file system in order to reach the desired performance.
The ideal interface for HPC yet has to be designed. Some suggestions have been made in this thesis, which should be considered when designing an interface specifically for applications in HPC. Some of these include metadata handling, making sure that only information is provided which is essential for HPC applications, and relaxed semantics, which should for example allow concurrent accesses to the same files if there are no overlapping write requests. Additionally, it should be possible to easily change I/O configurations in order to fit the individual access pattern of an application, as it is possible with ADIOS, thus allowing a number of applications with different access patterns to achieve good performance while using the same interface. Keeping all of this in mind when designing an interface which specifically serves the needs of high performance computing poses a challenge to future work.

Bibliography

[CST+11] Chen, Yong; Sun, XianHe; Thakur, Rajeev; Roth, Philip C. ; Gropp, William D.: LACIO: A New Collective I/O Strategy for Parallel I/O Systems. In: *Proc. of the 25th IEEE Int'l Parallel and Distributed Processing Symposium* (2011), May

[GWRG06] Grider, Gary; Ward, Lee; Ross, Rob ; Gibson, Garth: A Business Case for Extensions to the POSIX I/O API for High End, Clustered, and Highly Concurrent Computing. (2006), May

[HNH09] Hildebrand, Dean; Nisar, Arifa ; Haskin, Roger: pNFS, POSIX, and MPI-IO: A Tale of Three Semantics. In: *Proceedings of the 4th Annual Workshop on Petascale Data Storage* (2009), S. 32–36

[IBM] `http://www.almaden.ibm.com/StorageSystems/projects/gpfs [last checked: 2011-12-03]`

[Inc02] Inc., Cluster File S.: Lustre: A Scalable High-Performance File System. (2002). `http://www.Lustre.org/docs.html [last checked: 2011-11-30]`

[LKS+08] Lofstead, Jay; Klasky, Scott; Schwan, Karsten; Podhorszki, Norbert ; Jin, Chen: Flexible I/O and Integration for Scientific Codes Through the Adaptable I/O System (ADIOS). (2008)

[LRT04] Latham, Rob; Ross, Rob ; Thakur, Rajeev: The Impact of File Systems on MPI-IO Scalability. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (2004), S. 87–96

[PJS+94] Pawlowski, B.; Juszczak, C.; Staubach, P.; Smith, C.; Lebel, D. ; Hitz, D.: NFS Version 3 Design and Implementa-

tion. In: *Usenix Conference Proceedings (Boston, Massachussetts)* (1994), June, S. 137–152

[PSB+00] Pawlowski, Brian; Shepler, Spencer; Beame, Carl; Callaghan, Brent; Eisler, Michael; Noveck, David; Robinson, David ; Thurlow, Robert: The NFS Version 4 Protocol. In: *Proceedings of the 2nd International System Administration and Networking Conference (SANE2000)* (2000), May

[PTB+00] Prost, Jean-Pierre; Treumann, Richard; Blackmore, Robert; Hartman, Carol; Hedges, Richard; Jia, Bin; Koniges, Alice ; White, Alison: Towards a High-Performance and Robust Implementation of MPI-I/O on Top of GPFS. In: *Euro-Par* (2000), S. 1253–1263

[PTH+01] Prost, Jean-Pierre; Treumann, Richard; Hedges, Richard; Jia, Bin ; Koniges, Alice: MPI-I/O / GPFS, an optimized implementation of MPI-I/O on top of GPFS. In: *Proceedings of Supercomputing 2001*, 2001

[SGK+85] Sandberg, Russel; Goldberg, David; Kleiman, Steve; Walsh, Dan ; Lyon, Bob: Design and Implementation of the Sun Network Filesystem. In: *Proceedings of the Usenix Conference* (1985), June, S. 119–130

[SH02] Schmuck, Frank; Haskin, Roger: GPFS: A Shared-Disk File System for Large Computing Clusters. In: *Proceedings of the 2002 Conference on File and Storage Technologies* (2002), S. 231–244

[TGL98] Thakur, Rajeev; Gropp, William ; Lusk, Ewing: A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In: *Proceedings of SC98: High Performance Networking and Computing* (1998)

[TGL99a] Thakur, Rajeev; Gropp, William ; Lusk, Ewing: Data Sieving and Collective I/O in ROMIO. In: *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation* (1999), S. 182 – 189

[TGL99b] Thakur, Rajeev; Gropp, William ; Lusk, Ewing: On Implementing MPI-I/O Portably and with High Performance. In: *Proc. of the Sixth Workshop on I/O in Parallel and Distributed Systems* (1999), S. 23–32

[Top] `http://www.top500.org/list/2011/11/100 [last checked: 2011-12-03]`

[VLR+08] Vilayannur, Murali; Lang, Samuel; Ross, Robert; Klundt, Ruth ; Ward, Lee: Extending the POSIX I/O Interface: A Parallel File System Perspective. (2008), October

Appendix A

Application runtimes

n	ppn	POSIX I/O		MPI-I/O		ADIOS		
		NFS	tmp	non-coll.	coll.	POSIX	MPI	NULL
1	1	318,72	82,83	311,44	328,62	108,46	118,28	83,15
1	2	346,69	53,56	291,72	311,86	125,85	97,63	52,05
1	3	265,44	37,67	264,49	294,05	102,27	57,97	39,55
1	4	272,99	26,49	297,01	293,26	42,30	76,30	26,96
1	5	271,67	21,51	267,25	281,66	96,75	54,92	25,00
1	6	294,53	17,39	289,94	289,99	61,45	102,34	18,91
1	7	277,22	16,58	264,74	302,44	124,05	64,44	16,73
1	8	287,50	14,14	291,31	303,35	31,12	83,19	14,34
1	9	298,22	12,93	304,49	317,92	68,27	103,21	14,92
1	10	260,48	12,07	279,07	273,34	73,21	98,71	20,32
1	11	292,60	11,47	294,33	281,49	51,34	86,68	11,97
1	12	284,51	10,43	277,45	321,56	62,16	74,43	12,08
5	1	281,15	-	302,40	297,33	79,89	90,59	19,19
5	12	613,13	-	610,22	660,76	271,40	79,15	3,17
10	1	283,55	-	390,56	331,09	43,51	78,39	9,91
10	12	854,42	-	817,14	854,44	459,80	358,23	2,67

Table A.1.: Runtime in seconds using NFS. Application with I/O: 200 interlines and 500 iterations

n	**ppn**	**MPI-I/O**		**ADIOS**
		non-coll.	coll.	MPI
1	1	142,74	142,25	120,95
1	2	120,57	119,95	98,10
1	3	100,43	101,33	86,92
1	4	91,42	91,71	72,62
1	5	86,90	86,58	66,09
1	6	84,86	84,80	61,62
1	7	88,23	89,13	56,68
1	8	90,43	90,33	56,76
1	9	91,62	91,75	52,36
1	10	94,44	95,02	53,44
1	11	96,04	96,35	53,28
1	12	98,23	100,42	53,24
5	1	73,30	76,59	56,40
5	12	88,73	86,29	43,30

Table A.2.: Runtime in seconds using PVFS. Application with I/O: 200 interlines and 500 iterations

List of Figures

2.1 General layout of a distributed file system 5
2.2 The Network File System 7
2.3 GPFS Architecture 11
2.4 Lustre Architecture 13

3.1 An example for POSIX semantics 17
3.2 Session semantics, figure 1 18
3.3 Session semantics, figure 2 18

4.1 ADIO design 24
4.2 ADIOS architecture 25

5.1 Local file view 30
5.2 Global file view 31
5.3 Distributed array access 32
5.4 Data sieving 35
5.5 Collective read with Two-Phase 36
5.6 LACIO, figure 1 41
5.7 LACIO, figure 2 41

6.1 Comparison of throughput: NFS and PVFS. All processes running on one node 55
6.2 Comparison of throughput: NFS and PVFS. Processes running on five nodes. 55

List of Listings

4.1 Example XML file for ADIOS 26

6.1 writeCheckpoint function using MPI-I/O 47
6.2 writeCheckpoint function using POSIX I/O 48
6.3 writeCheckpoint function using ADIOS 49

Erklärung

Ich versiche, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel - insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen - benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 05.12.2011 .....