Fachhochschule
Bonn-Rhein-Sieg
University of Applied Sciences

Fachbereich Informatik
Department of Computer Science

Master Thesis

im Studiengang
Course of Studies
Master of Science in Computer Science

Optimization Strategies for Client-Server Mapping and Data Distribution Schemes in a Parallel Memory File System

by
Jan Seidel

Erstbetreuer: Prof. Dr. Rudolf Berrendorf
First Advisor

Zweitbetreuer: Prof. Dr. Peter Becker
Second Advisor

Eingereicht am: 28.02.2007
Handed in at

Abstract

This thesis addresses the storage of application data in parallel file systems. With the ever-increasing compute performance of parallel systems like clusters and supercomputers, increased amounts of data are processed. The size of this data often exceeds the capacity of the local main memory, requiring out-of-core storage in a file system. Parallel file systems are used to provide a shared view on I/O resources for multiple application clients and provide parallel, concurrent access to the storage devices.

We analyze optimization strategies for parallel file systems that help to improve the I/O performance of common parallel applications. By evaluating different I/O pattern analyses of existing parallel scientific applications, we identify two critical parts that can be optimized to provide high-performance I/O to parallel applications. Those parts are the communication patterns between I/O clients and I/O servers and the file data distribution on the I/O servers. Both parts together define the transfer of file data in the parallel file system.

Potential optimization strategies for these critical parts are discussed and evaluated throughout this thesis. One combination is chosen for implementation in a parallel file system that stores file data in the main memory of remote compute nodes. Utilizing the optimized strategies in this parallel memory file system, we are able to clearly exceed the I/O performance of standard parallel file systems for common application types.

Contents

Chapter 1

Introduction

1.1 Motivation

Parallel systems have shown a huge performance increase over the last decades. Projects like the list of the "Top 500 Supercomputer Sites" [1] show that the performance growth rate of these supercomputer systems nearly stays constant over the years [2]. This increasing performance allows parallel systems to process problems of increased size. Larger problems usually process more input data and create more result and temporary data. This data needs to be transferred from and to the underlying storage devices. The I/O performance growth rate of traditional storage devices like hard disks, however, can not keep up with the pace of processing performance [3, 4, 5], generating an ever-increasing gap between processing performance and I/O performance of a parallel system. This means, parallel systems can generate data more rapidly than this data can be stored by common I/O devices. This results in an I/O bottleneck, where the I/O subsystem is limiting the overall application performance. The standard access to I/O resources is serialized, for example with the Network File System NFS [6]. According to Amdahl's law, however, serialized parts of an application can significantly reduce the parallel scalability [7].

The most obvious solution to overcome this performance gap is to use parallel I/O. It uses multiple paths to multiple I/O storage ressources. By utilizing multiple paths to data, applications can access the storage ressources in parallel, increasing the overall achievable I/O performance.

Parallel file systems support this approach, mainly by distributing file data among multiple storage resources. This can increase the achievable I/O performance up to the product of the number of paths to different I/O devices and the performance of

a single I/O device. This standard approach of parallel file systems can become a bottleneck again, when the file distribution parameters do not fit the access schemes of applications. A typical I/O device delivers best performance when accessing the data in large contiguous blocks. A mismatch between the distribution scheme of the parallel file system and the access scheme of an application, however, can generate many small requests to the I/O devices.

Another potential bottleneck is the communication of file system servers and clients. When clients need to communicate with servers that do not hold the requested data, the performance decreases. Furthermore, the already mentioned mismatch of distributed and accessed data can also generate many small messages between clients and servers, while networks deliver best throughput for large messages.

In some parallel file systems there already exist solutions for these potential bottlenecks. We analyze and compare these solutions and present optimized approaches for parallel file systems in general. We are able to show significant performance increasements by using these optimizations.

1.2 Goals and Targets

This master thesis presents optimization strategies for parallel file systems. The thesis analyzes two basic parts of a parallel file system: The mapping of application clients to I/O servers and the distribution scheme of file data between these servers. The mapping defines which server is contacted for I/O requests by each client over the network, while the data distribution manages the storage of file parts on multiple servers. Taken together, these two parts define the access of applications to I/O data.

We developed a parallel file system that stores file data in the main memory of remote cluster nodes. The suggested optimization strategies will be evaluated in this parallel memory file system. It currently utilizes two basic approaches for the client-server mapping and the data distribution. The sophisticated techniques discussed in this thesis will be compared to these basic approaches to evaluate the achievable performance increasements.

Using this optimization strategies we show a significant improvement in the performance of typical parallel scientific I/O applications.

1.3 Roadmap of Thesis

Chapter 2 introduces the general architecture of parallel I/O systems, with high-level interfaces, the parallel file systems and the underlying storage hardware. It describes different file system architectures and shows the current state-of-the-art. In the following chapter 3, our own development, the parallel file system MEMFS is introduced with special respect to the design criteria that differ from existing file systems. Chapter 4 describes the access patterns of scientific applications, which facilitates the definition of requirements for an optimized parallel file system. Following this, we present an I/O cost model in chapter 5 that helps to measure the I/O performance of different file systems and optimization techniques. The following two chapters present optimizations for two selected parts of a parallel file system: The mapping of clients to servers in chapter 6 and the distribution of file data between I/O servers in chapter 7. Chapter 8 discusses potential combinations of client-server mapping and data distribution optimization techniques. Based on this comparison, a combination is chosen, that is implemented in the parallel memory file system MEMFS. Chapter 9 describes this implementation and discusses the achieved results. The last outlines the most important conclusions of this thesis and gives an outlook on future work.

Chapter 2

Parallel I/O Systems

This chapter introduces parallel I/O systems in general and presents some of the most widely used parallel file systems. The frequently used abbreviations of this and the following chapters are only defined once and can be found in the nomenclature prior to the bibliography of this thesis.

According to the definition in [4], a parallel I/O system shows three main characteristics:

- It stores file data on multiple storage devices,
- utilizes multiple connections between storage devices and compute resources, and
- provides high-performance concurrent access to the storage devices for multiple compute resources

By guaranteeing these three characteristics, multiple compute resources can access the storage devices through multiple connections in parallel. The third characteristic clarifies the difference to *distributed* I/O systems. Distributed systems also utilize multiple connections and storage devices, but are not designed for high-performance concurrent access to these. Distributed file systems often serialize accesses to the shared resources. NFS [6] for example is a distributed file system that uses serialization of accesses. In this thesis NFS is not discussed further, because it is not optimized for high-performance I/O.

A parallel I/O systems consists of multiple layers, as illustrated in figure 2.1. The lowest layer encapsulates the storage hardware, i.e. the storage devices like disks and the interconnects that are used to transfer file data. This layer also determines the

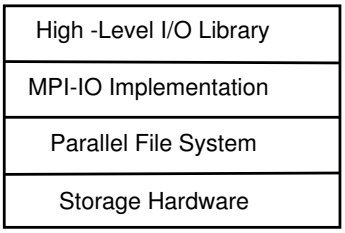


Figure 2.1: Parallel I/O Stack. Source: Based on [4]

maximum reachable I/O performance of the whole I/O system, by the performance parameters of the storage devices and interconnects. The parallel file system operates above that layer. It manages the data on the storage hardware, organizes data in files and directories and coordinates the access to files [4]. Most parallel file systems support the usage of RAID systems at storage hardware layer for improved reliability. The most common mechanism to improve performance in a parallel system is to use *striping* in the file system layer. Striping means to distribute file data blockwise to multiple I/O devices or I/O servers. It allows to leverage multiple servers, disks and network links during concurrent I/O operations of an application to shared files [8]. The application clients can concurrently access the file system through different connections. Parallel file systems have to coordinate accesses by multiple application clients to data, which can reintroduce bottlenecks [8]. Coordination is required to provide a shared and consistent view on the file system data for an application processed by multiple clients. Most parallel file systems use locking subsystems to ensure consistency of the stored file data. These subsystems lock files or parts of files to keep the file system in a consistent state. A sophisticated design of the parallel file system is required to minimize serialization of requests and facilitate parallelism in client accesses. File systems that cache data must furthermore especially ensure the consistency of caches.

The standard interface for parallel I/O is MPI-IO [9], which is a part of the standard programming interface for parallel applications, the *Message Passing Interface* MPI. This layer calls the parallel file system functions to access the I/O data. MPI-IO shows a great benefit in comparison to the standard Unix POSIX-interface [10] by providing structured data accesses to the application. MPI-IO translates accesses from the application into accesses that "can be performed efficiently on the underlying parallel file system" [4]. It also helps to decouple an application from a file system by providing an interface between the application and the underlying file system. A file system that implements the MPI-IO interface can be accessed by any application that uses MPI-IO operations. In the next section the MPI-IO architecture is presented in more detail. On the highest layer high-level I/O libraries

like "HDF5" [11] and "Parallel netCDF" [12] on top of MPI-IO provide more structured data accesses for clients. They allow an application programmer to directly pass multidimensional data types to I/O functions without the complexity without complex datatype definition as in MPI. The libraries translate these datatypes and structured accesses into MPI-IO accesses. These high-level I/O libraries map "application abstractions to a structured, portable file format" [8], which simplifies the development of high-performance parallel I/O applications. However, these libraries can add significant overhead to the application runtime [4], since the highest layer is implemented on top of MPI-IO and has to rebuild I/O functions in MPI-IO. An optimized implementation of an application that directly uses the MPI-IO operations can always reach at least the performance of the same application implemented with high-level I/O libraries. The following chapters of this thesis only consider the direct usage of MPI-IO operations, because most high-level interfaces are built on top of it.

The authors of [8] name the following three points as the main requirements of a parallel I/O system:

1. Provide mapping of application data into storage abstractions
2. Coordinate access by many processes
3. Organize I/O devices into a single space

The four layers of the parallel I/O stack work together as described above to fulfill these requirements.

2.1 The MPI-IO Interface

MPI-IO is a standard interface for parallel I/O, defined as a part of the MPI-2 standard [9]. There exist several implementations of the MPI-IO standard, the most commonly used one is ROMIO [13]. ROMIO is included in MPICH [14], Open MPI [15] and other MPI implementations. It implements the "abstract device interface for parallel I/O " ADIO [16] to create an abstraction layer of the underlying file systems. Any file system that implements the ADIO interface can be used by ROMIO for MPI-IO calls. ROMIO takes standard MPI-IO calls from an application and translates them in the ADIO layer into calls optimized for a specific file system. Apart from the standard I/O operations that exist in any I/O library, MPI-IO supports special I/O functions and constructs for parallel I/O. As these terms will be commonly used in the following chapters, they are briefly described here:

2.1.1 MPI datatypes

The MPI standard [9] introduces MPI specific datatypes. All MPI-IO operations that read or write data use MPI datatypes to describe the data in the I/O buffer. Datatypes are also used to describe the accessed data of a file, via the concept of file views, which is introduced in the next section. There exist *basic datatypes* for primitive machine datatypes. Some C examples are listed below with the corresponding C datatype in parantheses:

```
MPI_CHAR (char)
MPI_INT (int)
MPI_FLOAT (float)
MPI_DOUBLE (double)
```

Based on the basic datatypes the user can define new types. MPI provides several constructors to create *derived datatypes* consisting of multiple basic datatypes located either contiguously or noncontiguously. For example the *vector* constructor allows to build a datatype consisting of equally spaced copies of another type. The *darray* constructor builds a "process's local array obtained from a regular distribution of a multdimensional global array" [17].

A datatype built with a constructor can be used as the input for another datatype constructor, building nested datatypes. "Any noncontiguous data layout can therefore be represented in terms of a derived datatype" [17], especially nested strided data layouts. Strides of a datatype are holes in files or buffers that are skipped in accesses using this datatype. Using these constructors, complex data partitions of applications can easily be represented by MPI datatypes. With MPI datatypes, an application programmer is able to provide the whole access pattern of an application I/O stage in a single function call. This is necessary to fully utilize the advanced features of parallel file systems, as described in section 2.1.3. MPI datatypes are internally presented in a tree structure, expressing the nested structure. When accessing file data, this tree structure is traversed to skip the stride parts of a datatype.

2.1.2 File Views

Many applications define *file views* to express the accessed data of a file for each process. A file view describes the parts of a file that are visible to each process. File views are for example used to describe strided partitioning of files among multiple processes. By setting file views the processes can access their parts as if they

had a linear address space. This "relieves the programmer from complex index computation" [18], especially when accessing a file that is partitioned with nested strided patterns, where the computation of offsets can become very complex (see section 4.2). MPI-IO and many high-level I/O libraries support the usage of file views. In the MPI-IO interface, file views are defined by three parameters:

- A displacement, which is an absolute byte position, giving the location where the file view begins. This is especially helpful to skip file headers in views.
- The etype, which is the elementary "unit of data access and positioning within a file" [9]. Any MPI datatype (also a derived datatype) can be set as the etype of a file view (some restrictions apply, see [9]).
- The filetype "defines a template for accessing a file and is the basis for partitioning a file among processes" [9]. It can be the single etype or a construction of multiple instances of this etype. A filetype can contain holes, especially used for partitioning a file. The filetype can be constructed using the functions to create derived datatypes.

A view defines the data regions that can be accessed by a process. It begins at the defined displacement. The filetype is repeated over the file. The process can access all nonempty etypes of the filetype. See figure 2.2 for an illustration, where different file views are used for three processes to partition a file. Each process can only access the "filled" etype units of its filetype, which is repeated over the file.

File views are managed in the MPI layer. The three parameters (displacement, etype and filetype) are stored in the MPI_File object that is created when opening a file (see appendix A.2). The ADIO device of a file system uses these information to access the file data according to the view. When accessing file data, the datatype tree of the views' filetype needs to be traversed, writing or reading only those parts of the file that are accessible etypes of that view and skipping all other parts.

Views can also be seen as hints to the operating and file system. They define the accessible data regions of clients and can therefore help to optimize the "I/O scheduling, caching and pre-fetching policies" [18]. A file system can for example prefetch data for a client according to its' file view.

2.1.3 MPI File access

Applications access file data in special access patterns, defining which process reads or writes which part of a file. These access patterns can, however, be presented to

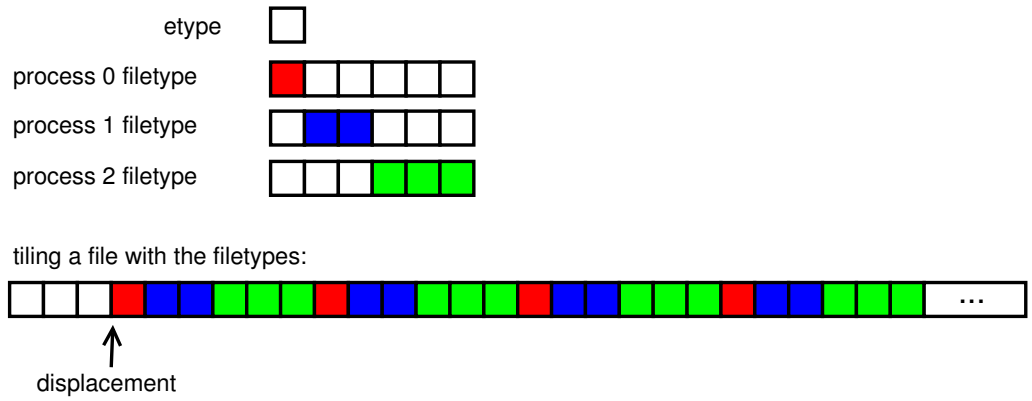


Figure 2.2: Partitioning of a file with file views. Source: Based on [9]

the file system in different ways. The authors of [17] classify four different "levels" of representing an access pattern in MPI-IO:

- The "lowest" level 0 equals the standard Unix-approach, where each contiguous part of a file is separately read or written by each process for itself. The stride between each contiguous part is then manually skipped by a *seek* operation. This level requires multiple I/O function calls for noncontiguous file data accesses.
- Level 1 extends the previous level by using the collective I/O functions provided by MPI-IO. Those functions indicate that all processes which opened the file call this function, each with its own request parameters. The file system can use optimizations for this collective case, as will be described in section 2.1.6.
- In level 2, each process defines a noncontiguous datatype and accesses all of the required file data with it in one independent (non-collective) I/O function call. This level reduces the number of I/O function calls and provides more optimization potential for the file system, as described in the upcoming section.
- Level 3 is similar to level 2 except that it uses collective I/O functions, providing both noncontiguous and collective optimization potential.

The four levels define increasing amounts of data per I/O request. The lowest level accesses each contiguous part of a request with a single I/O call for each process, while the highest levels accesses all file data in a single I/O call. The highest level also provides the highest optimization potential to the underlying file system. The performance improvement depends on how well the file system takes advantage of the increased access information [17]. To optimize an applications I/O performance, the programmer should always use I/O calls of level 3.

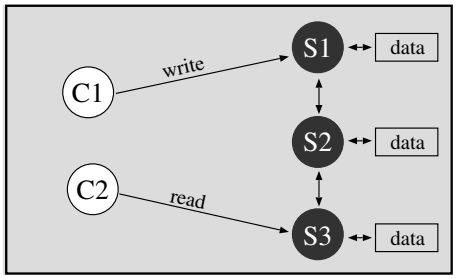


Figure 2.3: File consistency in distributed file systems

2.1.4 Sequential Consistency

The MPI standard [9] defines several conditions, especially the MPI-IO atomic mode, under which a file system must guarantee sequential consistency. Sequential consistency was introduced by L. Lamport in [19] as follows: "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program". For I/O operations this means that the results of a write operation must either be completely visible to all other operations or completely invisible. To further illustrate this, see figure 2.3, where client $C1$ performs a write operation to server $S1$ and $C2$ performs a read to $S3$. Assuming that these requests access overlapping file regions, the result of the read operation must either be the completely unchanged file data before the write operation or the completely changed data, but not a mixture of both. In parallel file systems this is a challenging task because the file system needs to avoid serialization of operations as much as possible to provide parallelism.

2.1.5 Noncontiguous I/O

A noncontiguous I/O operation is performed, when an application client accesses file data in pieces separated by gaps. MPI-IO provides sophisticated access operations to data that is read or written noncontiguously in many small pieces. There exist three forms of noncontiguous I/O: data that is noncontiguous in the memory, in the file or in both. The most common case is noncontiguous data access in files, which for example occurs when partitioning a file between clients. This partitioning is often done by using file views (see section 2.1.2). The concept of file views provides a "powerful way of specifying noncontiguous accesses" [20] of file data. Many real parallel applications partition file data among multiple processes. This will be discussed in more detail in chapter 4. The traditional way to "access noncontiguous

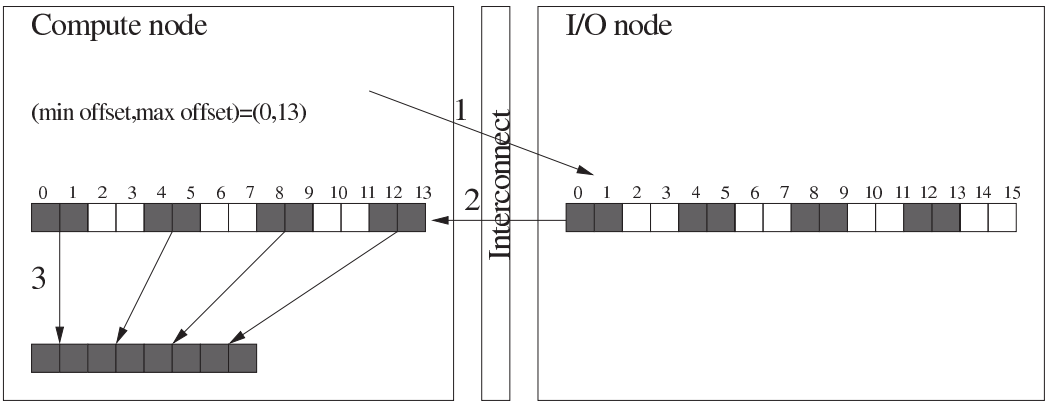


Figure 2.4: Data Sieving. Source: [18]

data is to use a separate function call to read/write each small contiguous piece" [4], as for example done by applications that use the POSIX-interface [10]. Because of the high latency of I/O devices, however, accessing many small parts of a file sequentially is very expensive. MPI-IO offers the ability to "access noncontiguous data with a single function call" [4]. Then, the ADIO device can call optimized functions of the underlying file system for noncontiguous data requests. If the underlying file system does not support noncontiguous I/O natively, the ADIO device can also use an optimization of ROMIO called *data sieving* [21] for I/O operations, illustrated in figure 2.4.

With data sieving, a read operation combines multiple noncontiguous I/O accesses into one request including the strides (step 1 in figure 2.4). This large request is performed, which "reads a single contiguous chunk of data starting from the first requested byte up to the last requested byte into a temporary buffer in memory" [21] (step 2). The data sieving algorithm then extracts the originally requested parts from this temporary buffer into the I/O buffer, skipping all strides (step 3). This reduces the impact of I/O latency by performing a single I/O operation instead of multiple calls. The extraction of contiguous parts from a noncontiguous buffer is performed in main memory, which has a much lower latency than disk-based devices (see [22]). However, the whole noncontiguous region including the strides is transferred over the network in step 2, increasing the transferred data amount.

Write operations can use data sieving with *read-modify-write* patterns [8]. The data reaching from the beginning to the end of the request is then first read into a temporary buffer, this buffer is modified according to the I/O request and then written back contiguously to the disk. To guarantee sequential consistency (see section 2.1.4) this write mechanism requires locking of the complete file region including the strides. Concurrent changes to the stride parts would otherwise be undone when the buffer

is written back to disk. Locking a whole contiguous part of a file, including the non-accessed strides of a request, introduces additional serialization of requests and reduces parallelism. ROMIO therefore provides a "user-controllable parameter that defines the maximum amount of contiguous data that a process can read at a time during data sieving" [21], limiting the locking to a region of this size. The parameter is also used to limit the memory-requirement of potentially large noncontiguous requests, which can span the whole file. Request sizes exceeding the value of this parameter are split into subrequests, which are performed consecutively.

2.1.6 Collective I/O

The data of parallel applications is often partitioned among multiple processes according to multidimensional distribution functions. This partitioning results in noncontiguous requests of single processes as described previously. The noncontiguous requests of different processes, however, "may together span large contiguous portions of the file" [21], for example when a matrix is partitioned among multiple processes. MPI-IO offers functions to describe collective I/O operations. The underlying ADIO device for a file system can then again either use file system specific functions for collective I/O or use ROMIO optimizations. ROMIO basically uses a technique called "two-phase I/O " [21] for collective I/O. Using two-phase I/O in the first phase multiple processes access file data in large contiguous chunks and in the second phase this data is distributed between the processes "to the desired distribution" [21]. This results in few, contiguous I/O accesses. This mechanism requires the exchange of buffers between processes but can often deliver better results than noncontiguous data access of individual processes (see performance measurements in [21]).

The special parallel I/O operations data sieving and two-phase I/O are "most useful when combined with a high-performance parallel file system" [4], as described in the upcoming section.

2.2 Parallel File Systems

This section briefly describes existing parallel file systems. The optimization for parallel file systems developed in this thesis will take solutions of existing file systems into account. Therefore, some of the most important parallel file systems are presented here. These file systems will be described with special respect to the dis-

tribution of file data and the mapping of clients to servers, the two optimization approaches this thesis analyzes.

Parallel file systems are designed for parallel access by multiple clients. Following the definition in [4] they can be grouped in two basic architectures: The *shared storage architecture* and the *intelligent server architecture*.

In a shared storage architecture clients access block devices on remote storage. The main goal is to "make blocks of [a] disk array accessible by many clients" [8]. Complete disk blocks are read and written by clients. This is either done directly through direct attached storage like a Storage Area Network (SAN) [23] or through intermediate I/O servers that provide block-oriented access to local storage devices, i.e. forward remote client requests to storage devices. Some shared storage file systems use virtual blocks for abstraction of the physical blocks to support data migration between servers and to provide a logical view of the directory structure of the whole distributed system [4]. The virtual block device furthermore provides a mechanism for adding or removing system hardware to or from the file system during runtime. Virtual blocks can be migrated to or from the added or removed hardware components. One drawback of the blockwise-access is the lack of native support for non-contiguous file accesses. All accesses have to be translated into contiguous block accesses on the client side before accessing the shared storage file system. Noncontiguous access patterns can result in complex "read/modify/write patterns that could have been avoided if more fine-grained accesses were allowed" [4]. Another drawback is that complete disk blocks are transferred between I/O clients and servers, even when only parts of these blocks are requested. This increases network load and makes false-sharing of blocks between different clients possible [8]. The servers are not designed to extract requested parts of blocks, they just transfer client request data to and from storage devices. A key component of shared storage file systems is the locking subsystem, which is utilized to coordinate access to the shared ressources [4].

In file systems that implement the intelligent server architecture the servers have more responsibilities than just providing blockwise-access to an I/O device. All communication with clients is done on a file and directory basis. This helps to support more complex and structured data requests like strided accesses, because the file system has more knowledge about what kind of data is accessed (for example a whole file, some noncontiguous part of a file or a directory). The real disk blocks are not visible to the clients, the file system is an abstraction layer from the concrete storage of data [8]. The actual disk blocks are read and written by the intelligent servers, so in case of partial block accesses only the requested parts are transferred,

reducing network traffic compared to shared storage systems. Intelligent server systems often separate the storage of *metadata* from the storage of file data [4], offering more flexibility in the number and placement of metadata servers. Metadata in file systems describes a file, "including owner, permissions, and location of data" [4]. It is required to maintain the state of the file system, including the directory structure. Some intelligent server file systems only support the usage of a single metadata server, because metadata operations require more coordination than file data operations. The single metadata server can become a bottleneck in large installations with many clients accessing it.

The main difference between the two architectures is that shared storage systems provide block-based access to a storage device and intelligent server systems provide file- and directory-based access.

Our development, the parllel memory file system MEMFS can be grouped into the latter architecture, because all requests are based on files and the servers provide functions to further describe requests.

Many parallel file systems were developed in the last years, some for scientific research, others for special parallel applications and again others for parallel I/O in general. As this thesis cannot describe all of these developments, it focuses on those systems that have shown the most significant impact on parallel file system architecture. The parallel file systems most widely used today are probably GPFS [24, 25, 26, 27] and PVFS2 [28], an advancement of PVFS [29]. Following these, there are other interesting developments like Lustre [30, 31] and GFS [32]. One very interesting parallel file system in terms of data distribution schemes is Clusterfile [18]. All of these are now briefly introduced.

2.2.1 Shared Storage Architecture

This section presents the parallel file systems GPFS and GFS with shared storage architecture.

General Parallel File System

The General Parallel File System (GPFS) is one of todays most widely used parallel file systems. It is a development of IBM and implements the shared storage architecture. It has especially shown its very high scalability and currently supports installations of up to 2441 nodes and a file system size of around 2 PetaByte [33].

It is utilized in many of todays fastest supercomputers (four clusters in the top ten list of the "Top 500 Supercomputer Sites" as of July 7, 2006 [33]) and has proven its very high performance in those installations.

Instead of an explicit virtual block device, GPFS can utilize a "Virtual Shared Disk" (VSD) component, which allows remote access to storage devices attached to multiple I/O nodes. Alternatively, the VSD component can be omitted if all clients are attached to a SAN and can directly access the storage devices. Then, all I/O traffic, including metadata, moves over the SAN [4]. In GPFS, one of the nodes accessing a file is designated as the *metanode* for that file [24]. This metanode is elected dynamically for each file. Usually this is the node that has opened the file for the longest time period. Only the metanode reads or writes the inode of a file from or to disk. The metadata is stored in this inode, containing file attributes and data block addresses. When a client needs to access a specific block it requests the data block address from the metanode of the file. The I/O transfer is then performed using this block address.

GPFS guarantees consistency by utilizing a distributed locking component [4]. This locking component coordinates the shared access to the disk blocks [8]. The granularity of this mechanism is at data-block level, which means that write operations to different data blocks of a file can proceed concurrently [4]. When a client writes data to a file it requests a lock for one or more blocks [24].

GPFS furthermore uses prefetching to exploit disk parallelism. Data is read into GPFS' buffer pool from as many disks as necessary to achieve the bandwidth of the switching fabric that the client uses to access the file data. GPFS recognizes sequential, reverse sequential and several forms of strided access patterns (see [24]). When recognized, data is prefetched into the buffer pool according to the access pattern of the client.

Global File System

The development of the Global File System [32] began in 1995 at the University of Minnesota. It was developed for large-scale computing clusters of the university, that generated huge datasets which had to be written effectively to a central storage [34]. GFS is now maintained and improved at Red Hat, where it was put under the GNU General Public License [32]. It uses a virtual block device architecture, the Logical Volume Manager (LVM). Storage devices are organized in *volume groups*, which are partitioned into *logical volumes*, the virtual equivalent of disk partitions. As in GPFS, nodes are either directly connected to shared storage or a component provides

remote access to storage devices over the Internet Protocol (IP) [4].

Data in GFS is stored as blocks on the virtual block device. GFS also uses a locking subsystem, OmniLock, to ensure consistency of file data. The locking granularity can be tuned to match application requirements and optimize system performance [4].

Red Hat GFS is used in many enterprise clusters to provide a consistent file system image over multiple nodes. There, it is deployed in environments like databases, application and web servers or high-performance compute clusters [32]. Those installations do not reach the sizes of supercomputer GPFS installations. GFS is currently limited to installations of about 300 nodes [35].

2.2.2 Intelligent Server Architecture

This section describes some of the most interesting intelligent server parallel file systems. It presents two file systems that show many design similarities, PVFS2 and Lustre and one file system which introduces a special way of distributing file data, Clusterfile.

Parallel Virtual File System

The Parallel Virtual File System PVFS2 [36] is a development of different scientific research groups, including the Parallel Architecture Research Laboratory at Clemson University and the Mathematics and Computer Science Division at Argonne National Laboratory. It is designed as a "high-performance scratch space for parallel applications" [4] and is also one of todays most-widely used parallel file systems. PVFS2 utilizes two types of servers, metadata servers and I/O servers that handle storage of file data. PVFS2 allows to distribute metadata among multiple servers removing a potential bottleneck of PVFS 1 in large installations. These metadata servers might also operate as I/O servers. The I/O servers store file data in the local file system of a node to reuse existing solutions for the basic storage task [8]. By doing this, PVFS2 does not need to implement a proprietary storage mechanism for file data and can rely on sophisticated techniques of existing file systems. PVFS2 usually distributes data by striping it among I/O servers. It also includes a modular system for adding new data distribution schemes to the system to support custom file data distributions. These distributions can be defined to match common access schemes of parallel scientific applications [28].

The access of file data by clients is based on *handles*. A handle is a large integer

that uniquely identifies an object stored in PVFS2. Handles exist for files, directories, symbolic links and also for the file partitioning elements, the data blocks stored at the specific servers. When accessing data, the filename is resolved into such a handle. The client furthermore performs all data accesses using this handle. PVFS2 supports the distribution of the handles among an arbitrary amount of servers. This is done by partitioning the handle space into ranges and delegating each range to a specific server. Clients receive information about the handle range distribution among servers at startup. A file name is resolved into a handle through a lookup operation [28]. File data in PVFS2 is split into *datafiles*, each datafile is also identified by a unique reference, the handle. When a client contacts the pvfs2-server responsible for that file for the first time, it receives information about the data distribution function used to partition the file among the pvfs2-servers. Using this distribution function, the client can compute the I/O server for each file byte. The distribution function is cached by the client for future requests. It can not be changed after opening a file, so no consistency mechanism is required for the distribution function.

The handle range concept furthermore makes it easy to add or remove pvfs2-servers during runtime: When adding a new server, a range of references (handles) is allocated for this server and added to the configuration table. When removing a server, its references are distributed to other servers and the table is updated. After the update, the clients need to be restarted with the updated table. This mechanism decouples the servers from the data stored on them. Currently, client-restart is necessary to update the configuration table, so updates are infeasible during the runtime of an application. The authors of PVFS2 are currently investigating the possibility to provide this functionality also during application runtime (see [28]).

PVFS2 does not provide POSIX consistency semantics or MPI-IO atomic mode semantics that guarantee sequential consistency as described in section 2.1.4. PVFS2 does guarantee atomicity of writes to nonoverlapping regions, even if these regions are noncontiguous. If application clients do not write to overlapping bytes, subsequent reads deliver consistent results [28]. The MPI-IO atomic mode needs support at higher levels of the parallel I/O system stack when using PVFS2. The authors of PVFS2, who also develop ROMIO, state that this will probably be done with enhancements to ROMIO [28].

Lustre

Lustre is a parallel file system developed by Cluster File Systems, Inc. Its active development started in 2002 and it is designed to provide high reliability, scalability

and performance [30]. Lustre was released as Open Source software under the GNU General Public License (GPL) [31]. It is a POSIX compliant file system designed for large installations [30].

Lustre provides only one metadata server (plus one backup server) that every client contacts when accessing a file for the first time. The metadata provides the clients with the I/O servers that handle these files, the so called *Object Storage Targets (OST)*. The client directly contacts these OSTs and exchanges I/O data with them, bypassing the metadata server completely. Although Lustre has been proven to be very scalable, the single metadata server can potentially become a bottleneck. For this reason, the authors are currently considering distribution of metadata information among the cluster (see [30]).

Lustre is an object-based parallel file system where all entities are represented as objects. Files are stored as objects on *Object-Based Disks*. This design especially supports the exchange of components with other components which provide the required functionality. The network abstraction layer (NAL) for example provides support for multiple types of networks, including TCP, Quadrics, Myrinet and InfiniBand [30, 37].

Lustre supports strong locking semantics to maintain consistency of the file system. The locking mechanism is distributed across the storage targets, each OST handles the locks for the object it stores [30].

Clusterfile

Clusterfile [18] is a parallel file system that is developed at the university of Karlsruhe. A typical Clusterfile installation consists of four main entities: a single metadata manager, multiple I/O servers, multiple cache managers and multiple clients as illustrated in figure 2.5. Clusterfile supports only one metadata manager, which can become a bottleneck in large installations that access a high number of small files. The I/O servers store file data in subfiles. These subfiles can be striped over multiple I/O servers or be kept at a single server. The servers utilize the cache of the local file system [18].

I/O clients can access Clusterfile through a kernel Virtual File Switch (VFS) interface and a user-level interface. On top of this user-level interface the developers implemented the MPI-IO interface, so that MPI-IO applications can be run with Clusterfile.

The developers of Clusterfile set special focus on the physical distribution of I/O data between servers. They state that "the performance and scalability of paral-

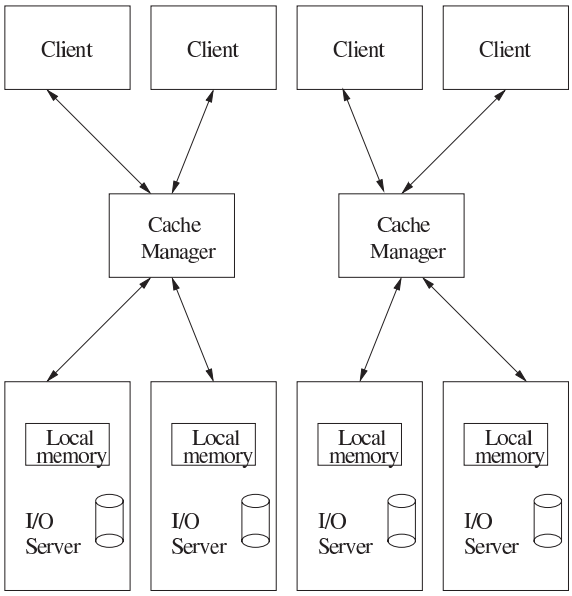


Figure 2.5: Clusterfile components. Source: [18]

lel scientific applications with intensive parallel I/O-activity suffer significantly from the mismatch between virtual partitioning and physical placement of file data" [38]. They furthermore argue that this is one of the reasons for "under-utilization of disk and network bandwidths" and "decreased parallel exploitation of independent disk capacity". Clusterfile therefore supports flexible physical partitioning of file data. The data distribution model is optimized for multidimensional array partitioning, which is very common in scientific applications, but supports arbitrary partitioning [18].

Clusterfile distinguishes between the logical partitioning (the file views) and the physical partitioning (the subfiles) of a file. The subfiles are not necessarily the identity of the file views. Clusterfile utilizes a common flexible data representation for both the physical and the logical data distribution. Both are represented by the so-called *nested* PITFALLS, which are an extension to the PITFALLS developed by Ramaswamy and Banerjee in 1995 [39]. Basically, nested PITFALLS describe nested strided segments of sequential buffers, but can express arbitrary distributions. They are a compact representation of complex regular distributions and there exist efficient algorithms for mapping and redistribution between two nested PITFALLS [39, 18]. With this extension, an arbitrary data partitioning can be described for both the physical and the logical data distribution. Obviously, this includes all MPI datatypes that can be used to define file views.

A file in Clusterfile is partitioned into multiple subfiles, where the partition elements are describing non-overlapping file regions and the union of all partition elements

describes a contiguous region. These two constraints guarantee that each byte of the file is mapped onto a unique partition element position. The single subfiles are then either written sequentially to an I/O node or striped over multiple I/O nodes, which is determined by the number of subfiles and the number of nodes. If there exist more subfiles than I/O nodes, each subfile is written to a single I/O node. The different subfiles are then distributed to nodes in a round-robin manner. If there are less subfiles than I/O nodes, the subfiles are striped on disjoint sets of I/O nodes [18]. The publication [18] does not mention consideration of special cases, for example when the number of subfiles is neither a multiple nor a whole-number divisor of the number of I/O nodes.

A view in Clusterfile is "a portion of a file that appears to have linear address" [18]. Views in Clusterfile have the same advantages as views in MPI-IO, described in section 2.1.2.

Chapter 3

Parallel Memory File System MEMFS

This chapter presents the parallel memory file system MEMFS. As described in the introduction, this thesis evaluates optimization strategies for parallel file systems. MEMFS is used to review these theoretical strategies in a running parallel memory file system.

The first section briefly introduces the motivation for the development of a new parallel file system, shows specifics of a memory file system and presents the whole project in which it is developed. To further motivate the development of MEMFS the following section describes special applications whose demands are hard to fulfill by currently available parallel file systems. Following this, the architecture of MEMFS is presented.

3.1 Project Overview

MEMFS is developed in the VIOLA project [40], which stands for "Vertically Integrated Optical Testbed for Large Applications in DFN". This project builds and evaluates a high-performance network testbed that connects different sites located across Germany. The core of VIOLA is the network that connects the project sites with dedicated 10 GBit/s optical WAN connections. Using this network, applications can be run on multiple clusters located at the project sites. The high-performance network also allows to couple remote clusters for applications with high data exchange between the processes. Parallel applications in VIOLA are run with MPMPICH [41, 42], a special MPI implementation for grid environments, that allows to

spawn MPI applications over multiple cluster sites. MP-MPICH is an enhancement of MPICH, it also includes the ROMIO parts for parallel I/O.

The development of a solution for high-performance remote parallel I/O in VIOLA is split into two parts: TUNNELFS [43] is used for transparent access to remote data in a grid. It implements an ADIO device that handles all MPI-IO operations of an applications client. As the name expresses, the TUNNELFS ADIO device tunnels the MPI-IO requests and communicates with remote I/O servers to fulfill the I/O requests of clients. This is done transparently to the user, especially no changes of the applications' MPI communicators are required. The I/O servers are special processes that are spawned by MP-MPICH. They can utilize all file systems that are supported on the server cluster as a target for the client operations.

MEMFS as the second part enables high-performance parallel I/O, comparable to any parallel file system, but with the characteristic that it completely stores file data in remote main memory. With MEMFS, data can be storead and retrieved from this additional remote storage with high-performance.

Together with TUNNELFS, MEMFS is utilized for high-bandwidth parallel I/O to the main memory of remote clusters in VIOLA. MEMFS operates without any disk I/O and therefore completely removes the bandwidth limitations of hard disks. Instead, it leverages the available memory resources of different cluster sites and is therefore able to satisfy very high bandwidth demands.

3.2 Use Cases for a High-Bandwidth Parallel Memory File System

The parallel file systems that have been described in the last chapter already show high parallel I/O performance when running on multiple I/O servers [44, 45]. However, there are several conditions under which these file systems are not that well-suited. A common parallel file system needs to be set up and configured on specific I/O nodes. This limits the usage of I/O nodes to exactly these preconfigured servers. Furtehrmore, deploying a parallel file system over several clusters can become a high administrative overhead, as this for example requires "common user bases and common security policies" [46]. This is still feasible in static grid setups, as demonstrated in the DEISA project [47] with GPFS. In reconfigurable grid environments like given in the VIOLA setup, however, not only clusters but also specific nodes of clusters are coupled dynamically on a per-job basis. Only these nodes are avail-

able for the application. This also requires dynamic distribution of server processes onto these dynamically coupled nodes. Such a setup requires a parallel file system that can be dynamically distributed among arbitrary cluster nodes. TUNNELFS and MEMFS provide such a dynamic parallel file system. This is probably the most important differentiating factor between this newly developed system architecture and other parallel file systems. TUNNELFS and MEMFS together are capable to create and terminate a parallel file system on an arbitrary amount of arbitrarily distributed I/O servers on a per-job-basis without prior setup.

The memory capacity of parallel systems is increasing rapidly [1, 48, 22] and reaches storage sizes that were reserved to secondary storage devices in the past. The result is that applications can keep more data in the fast main memory and avoid storage on relatively slow disk-based devices. But as already described, the problem size of typical parallel applications also increases, so secondary storage still remains necessary for most applications.

Nonetheless, applications can benefit from increased memory capacity of remote clusters. Some applications that frequently access large datasets are restricted to problem sizes that fit into the primary storage (see section 4.1.1). Other applications need to process data in real time, also requiring very low latency. These applications have very high I/O performance demands, so that access to secondary storage is often inapplicable if no high-performance I/O solutions are available. In this case, a parallel file system for main memory can help to increase the problem size. With high-performance parallel I/O to remote main memory the I/O performance can become sufficient to store frequently accessed data out-of-core. This means, not only problems that fit into the local memory of the application can be processed, but problems that fit into the aggregated local and remote main memory, increasing the processable problem size.

The general usage scheme of a parallel memory file system is to store temporary results, that are first written to storage and later read back. The constraint is that a memory file system can only store *write-read* data. It can neither store *write* nor *read* data, as defined below:

- Write data can be defined as result data, which should become persistently available after application termination. An application-based memory file system loses its data after job termination unless it is used as a caching system for a persistent file system. Currently, MEMFS does not provide this feature, but it is planned as an enhancement.
- Read data is input data, which is loaded into the application from persistent

storage. Before writing data to it, a temporary memory file system does not contain any data, so there is nothing that could be initially read from it.

- Write-read data is temporary data, that is first written by the application and later read back. This is usually intermediary result data, that is too large to be kept in local main memory. By introducing storage to remote main memory with MEMFS, this data does not need to be written to persistent disk storage, providing the high-performance of a memory-based parallel file system.

Write-read data only needs to be written to storage when the local main memory capacity is exhausted. A memory file system like MEMFS therefore is usually utilized to store data to remote servers. This means that in the most common cases application clients and I/O servers are placed on different clusters. This allows the application to use additional (remote) memory storage, without the requirement of deploying application clients on these remote cluster nodes. Using a temporary memory file system does not require any application changes, it just modifies the I/O storage target transparently to the user.

Many applications process more I/O data than can be stored in the available remote main memory, even when restricted to write-read data. Since a parallel file system for main memory provides relatively small storage capacity that can be accessed in high-performance, these applications can utilize it to store the most frequently accessed data to optimize the overall I/O time. By storing only the most commonly used files up to the overall memory capacity, the usage of the limited available remote memory is optimized. In this case, the parallel memory file system operates as an extended cache, that stores the most frequently accessed file data.

High availability of the file data and fault tolerance within the file system are not as important in temporary memory file systems as in persistent disk-based file systems. This results from the fact that file data is only available during job runtime anyway. If one of the nodes fails, jobs usually need to be restarted in a parallel system and the write-read operations are repeated. One approach to avoid a complete restart after system failure is to use *checkpointing*, but this is nontrivial in a parallel memory file system and is not a topic of this thesis.

Combining these factors, the main target of a remote parallel memory file system is to remotely store relatively small amounts of frequently accessed write-read data in dynamic grid environments. We discuss the special needs of parallel scientific applications where MEMFS could be employed in section 4.

3.3 Architecture of MEMFS

This section gives an insight into the design of MEMFS and its usage together with TUNNELFS [43]. MEMFS is started on some or all of the TUNNELFS I/O servers of an application. It does not require any administrative setup on the utilized nodes. TUNNELFS provides I/O server nodes that are spawned in the user-space as *extra processes* by MP-MPICH [41, 42]. These extra processes are hidden from the application. MEMFS can therefore be dynamically started on an arbitrary number of nodes in the grid [49].

MEMFS maintains a parallel file system for main memory on these nodes. The design of MEMFS can be divided in three parts:

1. The ADIO device,
2. the file storage, and
3. the multiserver environment.

These three parts are illustrated in figure 3.1, where the *main thread* and the *service thread* both belong to the multiserver environment. This is a simplified class diagram without any attributes and with only few of the I/O functions. The remaining functions are excluded for the sake of clarity. MEMFS does not support the standard POSIX-interface, because this interface is not designed for parallel I/O. For that reason, MEMFS can only be used via the MPI-IO interface. We now present the three MEMFS parts in more detail.

3.3.1 MEMFS ADIO Device

The MEMFS ADIO devices receives I/O function calls from the TUNNELFS server process, which originally received this request from a client process. The MEMFS ADIO device translates MPI-IO accesses into MEMFS-specific accesses and passes the parameters of the I/O operations to the locally running MEMFS server main thread. After the main thread finishes the handling of the I/O operation it returns the results to the ADIO device, which then also returns the results to the TUNNELFS server (see figure 3.2). A complete list of all MEMFS ADIO device functions can be found in appendix A.1.

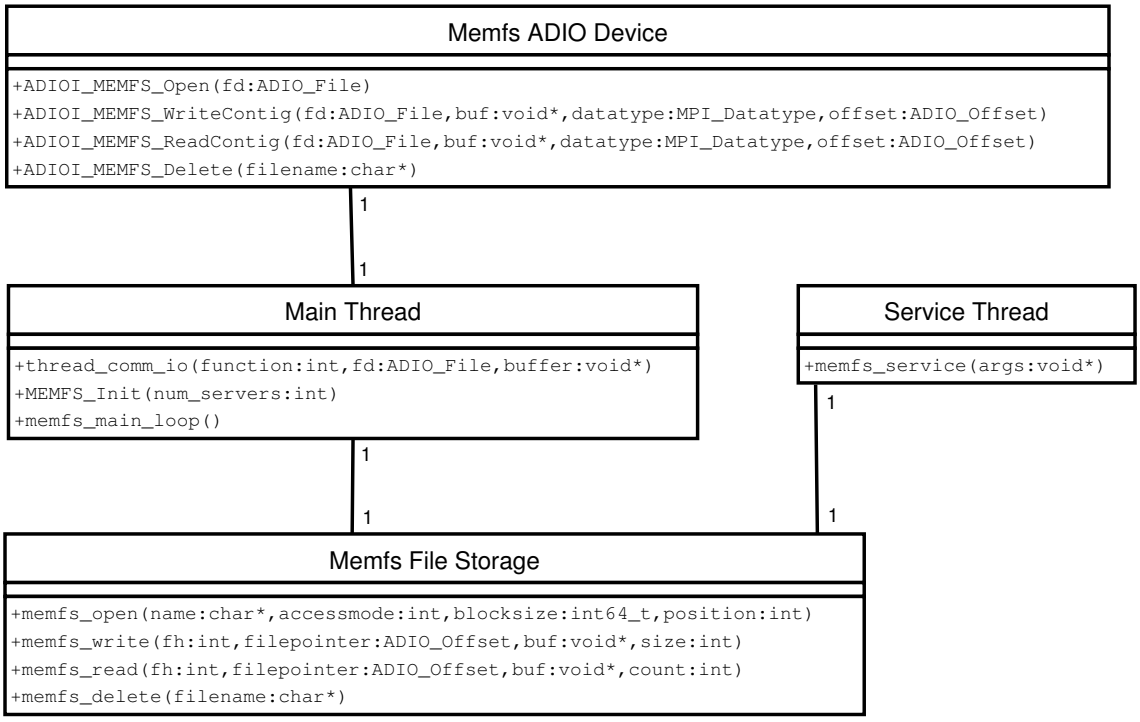


Figure 3.1: Class diagram of the MEMFS file system

3.3.2 File Storage in MEMFS

The file storage of MEMFS is designed to efficiently store file data in the main memory of I/O server nodes. This layer of MEMFS does not take charge of distributing file data among multiple servers, it stores all files sequentially in the main memory of the local server. The file storage consists of a filetable to administer the data and the I/O data itself. The filetable stores all metadata belonging to a file, such as the filename and the access mode. All metadata is distributed among the I/O server processes by creating the file on each server with the same parameters. The storage capacity of this local file storage is limited by the available main memory. In MEMFS, the I/O data is managed in blocks that are allocated and deallocated dynamically. The I/O *block size* can be set for each file by passing a parameter to the file system. If this parameter is not passed, a standard I/O block size is chosen. The file storage subsystem provides all necessary functions to execute MPI-IO operations, e.g. opening, closing, deleting, reading, writing and resizing a file.

3.3.3 Multiserver Environment

The MEMFS server nodes together form a parallel file system for main memory by communicating with each other. All communication in MEMFS (and also all

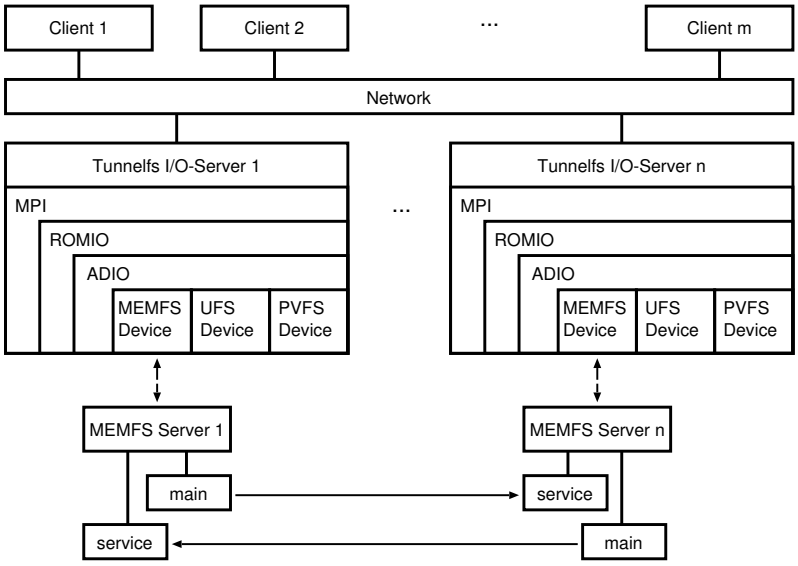


Figure 3.2: Interaction of I/O Clients, TUNNELFS Servers and MEMFS Servers

communication in TUNNELFS) is done with MPI operations. This helps to abstract from the underlying network device.

The multiserver part of MEMFS is responsible for the distribution of file data between servers. Currently, data in MEMFS is distributed by simple *striping* among all servers, which is one of the optimization approaches this thesis analyzes (see section 7).

The multiple MEMFS I/O servers are able to handle client requests concurrently, as necessary in a parallel file system (see section 2). Each server can handle the requests of any client. If a request accesses I/O data that is stored on other servers, these server communicates with each other and exchange the requested data. The communication of the different MEMFS servers is handled by two threads that are started on each server: the main and the service thread (figure 3.2). The main thread receives and handles all original requests from the ADIO device issued by an I/O client. It directly reads / writes data from / to the local file storage. If the I/O operation requires I/O data from one or more other servers, the main thread contacts the service thread of this / these server(s). The service thread of each server is responsible for the handling of requests from remote main threads. When the main thread received the results of all outstanding requests, it returns the combined result to the ADIO device, where it is returned to the calling TUNNELFS operation. This finishes the I/O operation in MEMFS.

3.3.4 The Locking Mechanism

MEMFS implements a central lock server to guarantee sequential consistency as introduced in section 2.1.4. The locking mechanism distinguishes between write and read requests. Multiple read requests to overlapping file regions can be performed concurrently, since no data is changed. When a write operation is active, however, no other write or read operations are allowed to concurrently access this file region. The locking server queues these requests until the write operation is finished. Upon finishing of a write operation, all queued requests are evaluated again to test if they can be performed or if further queuing is required due to other active write operations.

The locking mechanism of MEMFS is only required for applications with multiple clients that access overlapping file regions and if at least one of these requests is a write operation. The general access scheme of MPI applications, however, is often restricted to nonoverlapping file regions. Sequential consistency only guarantees that write requests are either completely visible or completely invisible, but it does not define any order on all requests, i.e. it does not define if a read returns the data before or after the changes of a concurrent write operation. This is too relaxed for many applications, which is the reason that accesses to overlapping file regions are often already negotiated at application level. Since the locking mechanism introduces additional overhead, applications that do not require sequential consistency guaranteed by the file system should disable it.

3.3.5 Optimized access patterns for MEMFS

MEMFS is currently optimized for large, contiguous accesses. Noncontiguous accesses are currently completely handled by ROMIO, which performs data sieving. The current implementation of MEMFS delivers high performance for optimized access patterns, but this performance collapses with noncontiguous accesses.

To optimize the performance of the parallel memory file system MEMFS we identified two essential parts: The client-server mapping and the file data distribution between servers. The client-server mapping defines the coupling of I/O clients and I/O servers. Optimized solutions for both parts are required to minimize communication between MEMFS servers and provide optimized, parallel access to an application.

The following chapters analyze, compare and evaluate optimizations for parallel file systems to provide high performance also for non-optimal access patterns.

Chapter 4

Access Patterns of Scientific Applications

To optimize a parallel file system, knowledge about the applications accessing it is required. Applications in general have very different I/O access schemes, so examining a wide variety of applications would yield multiple different access schemes. However, specific access types are very common for a wide range of applications. This chapter analyzes multiple scientific application with respect to these common access types. Several authors have already examined the I/O access patterns of scientific applications in the past, for example in [50, 51].

The CHARISMA project of N. Nieuwejaar et al. described in [50] has analyzed the usage of "multiprocessor file systems" in parallel applications used in real production environments. Unfortunately this project was finished in 1996, so these results are older than one decade now. Typical scientific applications changed since then, especially in terms of the amount of data processed, but some of the CHARISMA results are still very interesting. The project was started to "CHARacterize I/O in Scientific Multiprocessor Applications from a variety of production parallel computing platforms and sites" [50]. The work includes characterization studies on two systems: An Intel iPSC/860 [52] with the "Concurrent File System" and a Thinking Machines CM-5 [53] with the "Scalable File System". Both file systems are out of date and not used by todays clusters.

The I/O analysis of F. Wang et al. from 2004, described in [51], traces the system I/O activities of three parallel scientific applications: two physics simulations, one running on 343 nodes and one running on 1620 nodes and the I/O *Stress Benchmark ior2* [54]. The underlying file system used by the authors is a development version of Lustre Lite [30, 31]. The trace data was collected on a ASCI Linux Cluster of

the Lawrence Livermore National Laboratory, which is a 960 dual-processor cluster connected through a Quadrics Switch. This more up-to-date analysis can probably better clarify the I/O needs of todays' applications. The restriction to three applications, however, can not deliver results for parallel applications in general, but it can give some interesting insight into special parallel scientific I/O requirements.

Both works describe the I/O behavior of parallel applications with focus on the amount of files, the file sizes and the access schemes. We now outline the most interesting results for the optimization of a parallel file system.

4.1 Load Analysis

4.1.1 Out-of-core Applications

In CHARISMA, most of the examined applications opened only a few files during their execution - more than 75% opened less than 8 files - but a few applications opened many files (the maximum was 2217 opened files for one application). The authors conclude that although many applications only open a few files, "file system designers must optimize access to several files within the same job" [50].

Another result is that less than 5% of all files opened were "temporary", here defined as a file deleted by the same job that created it. Only few of the analyzed applications use files as an extension of memory for out-of-core solutions. The authors conclude that most "programmers have found that out-of-core methods are in general too slow" [50] due to the limited I/O performance.

In [51], no analysis of the number of files used by each applications is done. The authors only discuss that during their traces the file servers stored about 300.000 files altogether. From these files, 6.6% had a lifetime shorter than one day, which can be compared to "temporary" files. These files make up 7.3% of the stored data amount.

In a parallel memory file system all files are temporary, except this memory file systems supports storage at another persistent storage device. Files are only available during runtime to store intermediary results that do not fit into the main memory of the compute nodes. Only these relatively few files (5% and 6.6% respectively) can be opened with MEMFS, but if applications take advantage of the increased I/O performance, this number can increase. This can also help to increase the applications' problem size by providing high-performance access to out-of-core data.

4.1.2 File and Request Sizes

The results regarding file sizes or I/O request sizes described in CHARISMA are not discussed in detail, because these values are probably obsolete due to their age. It also not possible to just multiply the results by a constant factor because it is uncertain if file and request sizes of scientific applications were increasing proportionally since these results were taken. Anyhow, some basic facts of the CHARISMA results are described. An interesting point is that small request sizes (up to 1000 byte) were very common but still most of the data was transferred in a few large requests. The conclusion of the authors is that the analyzed applications require low latency for small requests as well as high bandwidth for large requests.

In the more current results of [51], the benchmark ior2 has one unique request size of around 64 kilobytes. In one of the physics simulations almost all write requests are smaller than 16 bytes while still most of the data is transferred in requests larger than one megabyte. In this application there occur nearly no read operations. The other physics simulation uses two major write sizes: 64 kilobytes and 1.75 megabytes. The read requests mostly have sizes less than 1 kilobytes, while still a few larger requests of 8 kilobytes make up 30% of all transferred data.

Taken these results together it becomes apparent that parallel scientific applications often use a large amount of small requests, while still most of the data is transferred in relatively large requests. This again shows the requirement of both low latency for small requests and high bandwidth for large requests. A high-performance parallel file system should be optimized for both of these two requirements.

4.2 Data Access Schemes

A large number of applications like video and audio streaming, copying of data files and archiving access a file sequentially from the beginning to the end. This is optimal for most underlying file systems, because "many systems can identify this pattern and optimize for it" [4], for example by reading complete disk blocks at once. However, parallel scientific applications often show other forms of access patterns.

The CHARISMA authors examined the access schemes of the running applications. They define a *sequential* request "to be one that begins at a higher offset than the point where the previous request from that compute node ended" [50]. The analyzed applications show that nearly 100% of the write-only files were accessed sequentially. This is because most of the analyzed applications wrote to separate

files for each node. With a separate file there usually is no need for a node to write a file in non-sequential order. Likewise most but a significantly lower percentage of the read-only files were accessed sequentially, because most read-files were shared among the nodes. Several of the applications that read files in a non-sequential order, read the data in reverse order, beginning at the end of the file, an access scheme called *reverse sequential*. Most of the read-write files were accessed non-sequentially. The authors give no specific explanation for that, but one can reason that the non-sequential skips occur especially when changing from a write to a read and otherwise.

Another important analysis of the CHARISMA project is the regularity of access patterns. Therefore, the authors analyzed the *interval* between requests, which is the "number of bytes between the end of one request and the beginning of the next" [50]. One third of the files was accessed in a single read or write request per node, so there was no interval at all. The remaining accesses show a high regularity. Only less than 10% of all files were accessed with more than two different intervals. The request sizes (the amount of requested data) show a high regularity, too. Less than 20% of all files were accessed with more than three different request sizes. The interval and the requests sizes together show that many applications used "regular, structured access patterns" [50]. The authors conclude that this is probably because much of the data was stored regulary in matrix form.

When analyzing the nonconsecutive access patterns, the authors found that many of the files were accessed entirely with a *strided* pattern. A series of requests is *simple-strided* if each request accesses the same amount of data and the interval is always the same between requests. A *strided segment* is a number of requests that are part of a simple-strided pattern. Then a *nested strided* access pattern is "composed of strided segments separated by regular strides in the file". This nested strided pattern occurs when multidimensional matrices are distributed among multiple processors. See figure 4.1 for an illustration of such a distribution, where a threedimensional matrix (a) is distributed among four processes. The strided access to the corresponding file is shown in (b). These nested strided patterns occur very often in scientific applications, because a lot of scientific data is stored in matrix form and the matrices are distributed among multiple processes. The authors of [4] call this "in some sense worst-case scenarios for parallel I/O systems", as these strides result in many small-sized requests that only read small parts of each disk block.

This indicates that applications "can benefit from the descriptive capabilities available in high-level interfaces" [4] as described in chapter 2. Programmers for example can use MPI derived datatypes to describe nested strided accesses in memory or in

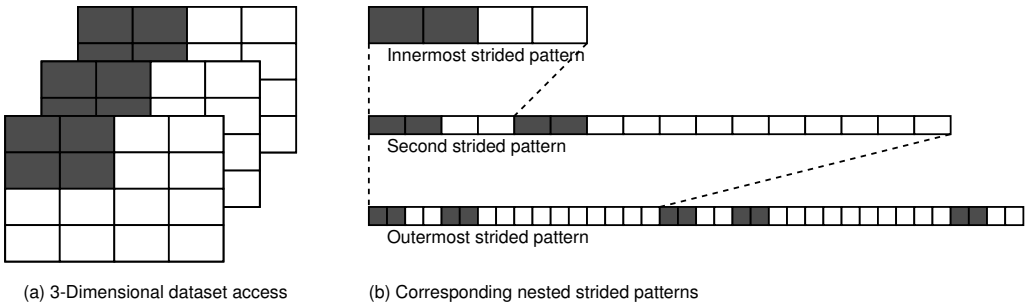


Figure 4.1: Nested-Strided Example. Source: Based on [4]

files (see 2.1.2). The frequency of these strided access patterns also show that the layers below the high-level interfaces should support operating in terms of structured data [4]. This means that noncontiguous file accesses should be natively supported by the underlying parallel file systems.

The authors of [51] also utilize a strided version of the ior2-benchmark [54]. This version strides the "blocks from different nodes into the shared file" [51]. In this strided version, the write performance of 512 nodes drops down to 2 gigabytes per second, compared to 9 gigabytes per second in the contiguous version of the benchmark. The read performance even drops down from about 4 gigabytes per second to 100 megabytes per second. [51] The development version of Lustre Lite used in this project does not natively support strided access patterns which results in this drastical performance decrease.

From this results we can conclude that a parallel file system which natively supports and is optimized for nested strided access patterns can greatly improve the performance of many parallel scientific applications. A file system that stores file data according to the distribution of data between application clients can access the storage devices in contiguous large chunks.

4.3 Implications for the Design of a Parallel File System

The two access pattern characterizations described in this chapter have shown some requirements for a parallel file system that can help to improve the I/O performance of typical parallel scientific applications. We will keep these in mind when discussing potential optimizations for parallel file systems. We identified three important requirements, that a file system should support to achieve optimized results:

1. A file system needs to deliver high performance for a large amount of files,
2. it should provide low latency for small request sizes as well as high bandwidth for large request sizes and
3. it should be optimized for nested-strided access patterns

Chapter 5

I/O Performance Optimization

The main goal of this thesis is to optimize a parallel memory file system for common parallel scientific applications. The total I/O time t of an application should be optimized. This is the total amount of time that all I/O operations of an application take. In a parallel file system there are two main factors that influence the total I/O time: the network transfer of data and the processing of data on servers, including the access of the I/O devices. This chapter defines a general mathematical cost model that helps to evaluate the optimization strategies introduced in the upcoming chapters. This simplified cost model only defines the costs for single clients, not for the total amount of application clients. This is a simplification of the real I/O time, because clients operate in parallel and concurrently, so the total costs of an application can neither be computed by just adding up all client costs nor by taking the maximum of the client costs. This simplification is necessary to keep the cost model at an expedient complexity.

The model constrains to setups where an application runs on two coupled clusters. Let this be cluster $Cl1$ and cluster $Cl2$. In the cost computations for a parallel memory file system it is furthermore assumed that all I/O clients are deployed at $Cl1$ and all I/O servers at $Cl2$ (see section 3.2 for an explanation why I/O servers and clients usually are not placed on the same cluster). This setup is illustrated in figure 5.1.

The nodes of each cluster are connected through an internal network interface. The parameters of this network interface are: The latency of sending a message from one node to another node of the same cluster: l_{Cl1} and respectively l_{Cl2} and the bandwidth of the internal network connections between the nodes: b_{Cl1} and b_{Cl2}. Then, let there be a intercluster network device I connecting $Cl1$ and $Cl2$. This device is described by a latency l_I and a bandwidth b_I. Furthermore each message traverses

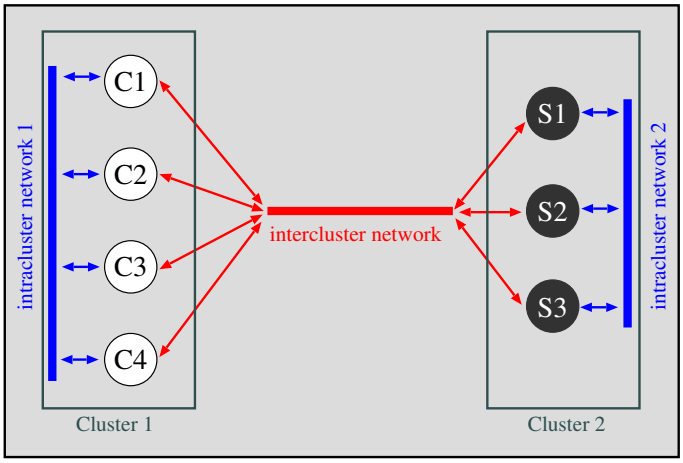


Figure 5.1: Setup for 2 Clusters: All Clients are placed on Cluster 1, all Servers on Cluster 2

a number of hops. These hops are only modelled on the server cluster $Cl2$. On $Cl1$ the clients directly contact the I/O servers for each I/O request. There are situations where data needs to be internally sent over multiple hops of the client network, for example when only a subset of clients is connected to the intercluster network. But these hops are hidden to the file system layer and can therefore be included in the bandwidth and latency parameters of $Cl1$. The servers, however, may need to actively communicate with other servers to exchange the requested data of the parallel file system. The potentially multiple hops at the server-cluster are defined as: h_{Cl_2}. There are also no hops associated to the parallel file system on the intercluster network. The file system layer directly exchanges data between I/O clients and servers, not regarding any routers or other hops of the network in-between. Therefore, the cost model does not include hops at the intercluster network. The routing of messages over the intercluster network is completely determined by underlying protocols. These costs are contained in the bandwidth and latency parameters of the intercluster network.

Then, the transferred I/O data itself is influencing t: Let there be an ordered set of messages

$$M_j = (m_{j1}, m_{j2}, \ldots, m_{jn})$$

for client j. The number of messages sent and received by client j is

$$n_j = |M_j|$$

and let the amount of data transferred per message $m_{ji} \in M_j$ be d_{ji}.

The data needs to be processed on each hop. It is assumed that the processing time of a message is identical on each hop, which again is a simplification. Furthermore

it is assumed that the processing time is somehow related to the size of the message, so the processing time pt is defined as a function c of d_{ji} and a constant factor tc, which is the time that a hop needs to process a message, regardless of the size:

$$pt = c(d_{ji}) + tc. \tag{5.1}$$

The transfer time of a message m_{ji} between two nodes of cluster $Cl2$ can now be defined as:

$$a_{Cl2} = h_{Cl2} \cdot (l_{Cl2} + (d_{ji}/b_{Cl2}) + pt) \tag{5.2}$$

The transfer time of the intercluster network can be defined in the same way, except that no hops are modelled:

$$a_I = l_I + (d_i/b_I). \tag{5.3}$$

The I/O time t_{ji} for each message $m_{ji} \in M$ is:

$$t_{ji} = a_I + a_{Cl2}. \tag{5.4}$$

Requests exceeding the maximum network message length, require are split of the I/O data into multiple messages. TUNNELFS, for example, currently uses a maximum message length of 4 MB, I/O data requests exceeding this value have to be split up, generating multiple I/O data messages.

This cost model is used to analyze the MPI-IO operations of scientific applications, that access parallel file systems. It concentrates on intelligent server architecture file systems as this is the most common file system design, that is also used in the parallel memory file system MEMFS. In these file systems an I/O operation can be divided into several stages. The steps of a write operation are described now, as illustrated in figure 5.2:

1. The client application on cluster $Cl1$ calls the MPI-IO operation.
2. The file system specific ADIO device is called by MPI-IO.
3. If necessary, this ADIO device splits the request into parts, either because of message size limits, or to directly contact multiple servers (compare to the mechanism of PVFS2 described in section 2.2.2), or both.
4. The ADIO device initiates a write operation with one or more I/O servers running on $Cl2$ (one or more network messages).

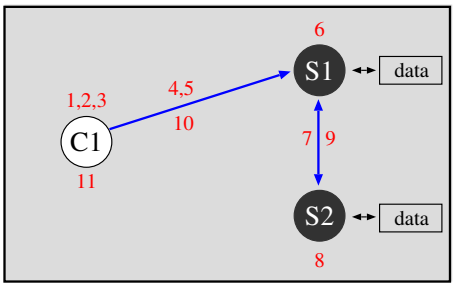


Figure 5.2: Different steps when an I/O function is called

5. It then sends the individual data parts to this server / these servers (one or more network messages).

6. Each contacted server receives these parts.

7. If required, the server forwards some or all of the parts to one ore more target servers (zero or more network messages).

8. The target servers receive and write the parts into the target file system.

9. The target servers and the originally accessed server exchange error codes (zero or more network messages).

10. After writing all parts and exchanging all error codes with target servers, the originally accessed server sends an error code to the ADIO device (one network message).

11. The ADIO device analyzes this error code and returns it to the MPI-IO application.

Taken together, one I/O request results in multiple messages exchanged between clients and servers, at least three messages for each I/O request, potentially more.

A read operation performs similar, except that the I/O blocks are sent from the servers to the client. In this case, the client initiates the data transfer by a small request message sent to one or multiple servers.

All exchanged messages can be modeled with the cost model described above. The total time for a specific I/O request is then the sum of the time taken for all these messages.

The total I/O time for a client j can be computed as:

$$t_j = \sum_{i=1}^{n_j} t_{ji} \tag{5.5}$$

This is a very simplified model because it does not regard potential network optimizations like pipelining messages or conflicts like collision of messages. This model can be used to analyze and improve the I/O time for specific clients. However, the target of a parallel file system is to optimize the total I/O time of applications and not the I/O time of single clients. Therefore, the cost model can help to optimize a parallel file system, but it can not be used as a simple formula, which just needs to be solved optimally. The following optimization approaches will be evaluated with the support of this model, but will not solely rely on it.

Chapter 6

Optimization Schemes for Client-Server Mapping

The first optimization technique for parallel file systems analyzed in this thesis is the mapping of clients to servers. A *mapping* is an assignment such that a client contacts one specific server for all MPI-IO requests until the mapping is changed.

Formally a mapping can be described as: Let there be a set of clients

$$C = \{c_1, c_2, \ldots, c_k\}.$$

Let there also be a set of servers

$$S = \{s_1, s_2, \ldots, s_l\}.$$

Then a mapping is a total function

$$f : C \to S, \tag{6.1}$$

defining one server for every client.

An important factor for optimizing the client-server mapping of an application is the *data matching* of client requests. It can be defined as the percentage of requested data located at the contacted server: Let a request R be described by an offset given in bytes off, a file view $view$ (see section 2.1.2 for a definition) and an amount of data to be read or written rs:

$$R = (off, view, rs).$$

A function

$$g : R \times S \to \mathbb{R} \tag{6.2}$$

with

$$g(r, s_i) = x, 0 \leq x \leq 1, r \in R, s_i \in S, x \in \mathbb{R} \tag{6.3}$$

computes the matching for a given request r and a server s_i. What g computes is the percentage of data requested in r that lies on server s_i. When file data is not replicated among multiple servers the sum of all server matchings is always 1:

$$\sum g(r, s_i) = 1, i = 1, \ldots, l. \tag{6.4}$$

If not otherwise stated we assume that file data is not replicated. A matching

$$(r, s_i), r \in R, s_i \in S$$

is optimal if there is no other matching $(r, s_j), s_j \in M$ with

$$g(r, s_j) > g(r, s_i),$$

i.e. s_i is the server that stores most of the data requested in r. Let *opt* represent this optimal server s_i, so $g(r, opt) = x$ is the optimal data matching for request r.

The more data a client directly reads from or writes to the assigned server, the less data has to be transferred between different I/O servers. The maximum reachable data matching for a request is the maximum percentage of requested data that lies on any one of the servers. If a client accesses in a request r a total of n bytes and $n/2$ bytes of that request are located at server $s_1 \in S$, again $n/2$ bytes are located at server $s_2 \in S$, then the maximum reachable data matching is 50%, assigning the client to either one of the servers s_1 and s_2. In this case, g is:

$$g(r, s_1) = g(r, s_2) = 0.5.$$

There is no unique optimal server for r, so in this case *opt* is defined as the set containing both s_1 and s_2 : $opt = \{s_1, s_2\}$.

The only possibility to increase this value is by dividing the original request into requests for specific servers: Assume that r' contains all data located at server s_1, while r'' contains all data located at server s_2: $r' + r'' = r$. Then

$$g(r', s_1) = g(r'', s_2) = 1.$$

In this way, a data matching of 100% can be achieved, as no intraserver communication is necessary to transfer the requested data from / to the client. Approaches that split requests and directly access the right servers for each part of the request will be described in section 6.2.1 and 6.2.2.

6.1 Current State in MEMFS

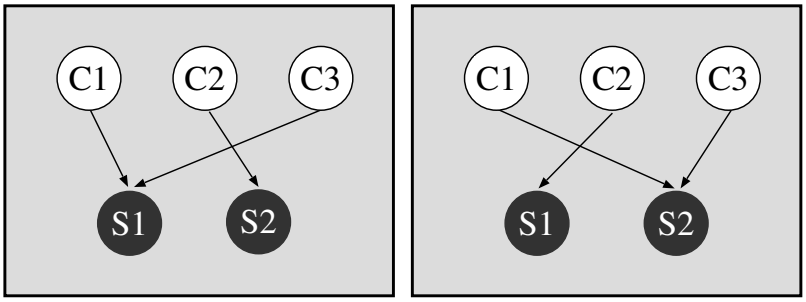
6.1.1 Client-Server Mapping Interface

As described in chapter 3, MEMFS does not directly communicate with I/O clients. All client requests are handled by the TUNNELFS client, which forwards these requests to the I/O servers and then calls MPI-IO operations itself. So MEMFS can not directly change the mapping of clients to servers itself. It implements an interface that passes changes to the client-server mapping through the TUNNELFS server. TUNNELFS allows to change the mapping after each MPI-IO operation that is called by a client. The TUNNELFS server can add a new mapping as a parameter of the reply to the client. The client automatically evaluates all reply parameters and – if given – sets a new client-server mapping.

When using MEMFS as the target file system, the TUNNELFS servers call a MEMFS-specific function, which computes a mapping with the client-server mapping algorithm implemented in MEMFS. Currently, this is a simple round-robin mapping algorithm, which is introduced in the next section, but it can be exchanged with more sophisticated algorithms, which are presented later in this chapter. Currently, the function which computes a mapping is called at the end of each open, read, write and file view modification operation. But if required, the interface can be changed to call this function at the end of any MPI-IO operation. Mappings can only be changed at the end of an operation, because only then the new mapping can be returned as a parameter to the TUNNELFS I/O client. Advanced mapping algorithms that require changes in the mapping during an I/O operation (for different blocks of a write request for example) would require changes to the general parameter exchange procedure of this interface.

6.1.2 Round-Robin Mapping

Currently MEMFS uses a static client to server mapping. It uses a n-to-1 client to server assignment, which means that each client contacts exactly one server and receives all results from this server, while a single server can be contacted by multiple clients. In the current implementation, each client gets assigned one server in a round-robin fashion, which results in an even distribution of clients to servers for all files together. The mapping for a file starts, where the last mapping ended. This guarantees that the algorithm reaches a nearly even distribution of clients to servers. See figure 6.1 for an example where three clients are assigned to two servers. The



(a) Client-Server Mapping for File 1

(b) Client-Server Mapping for File 2

Figure 6.1: Round-Robin Client-Server Mapping for multiple Files

mapping of file 1 assigns two clients to server 1, the mapping of file 2 assigns two clients to server 2. So in total, each server gets assigned the same number of clients. In a round-robin mapping the mapping function f is defined as:

$$f(c_j) = s_i, i = (1 + x + j) \bmod |S|, s_i \in S, c_j \in C, \tag{6.5}$$

where x is the server number at which the last mapping ended. The constant value of 1 is added to this server number to start at the directly succeeding server of the last mapping. Servers are numbered in terms of their rank in the internal server MPI communicator. A round-robin assignment allows to parallelize requests coming from clients, as different clients contact different servers concurrently. This is one of the requirements of a parallel file system (compare to section 2).

In a *globalmaster* assignment in contrast, all clients are assigned to exactly one server, so parallelization of accesses is not possible. In this case, f is defined as:

$$f(c_j) = s, c_j \in C, s \in S, j = 1, \ldots, |C|. \tag{6.6}$$

This globalmaster mapping corresponds to a distributed file system like NFS, that supports sharing of files between multiple clients, but no parallelism in client requests.

A problem arises, when clients access data of other servers than the one they are assigned to. In this case, the server has to communicate with one or more other servers to read or write the requested data, increasing the number of hops of the server cluster by one. This obviously results in increased network traffic, slowing down the client request. The difference can be shown with the cost model developed in section 5: When a client directly contacts the server that holds the requested data, this server does not have to further communicate with other servers to read / write

the data. The number of hops at the server cluster $Cl2$ is then 1, which can be inserted into equation 5.2:

$$a_{Cl2} = 1 \cdot (l_{Cl2} + (d_{ji}/b_{Cl2}) + pt). \tag{6.7}$$

If the data matching is below 1, parts of the accessed data need to be transferred between servers. If the roundrobin mapping assigns the optimal server *opt* to a client for a request r, giving the data matching $g(r, opt) = x$, then $1 - x$ percent of the accessed bytes need to be transferred from a second server, increasing the number of hops on $Cl2$ to 2. The overall transfer time on $Cl2$ is then:

$$a_{Cl2} = x \cdot (l_{Cl2} + (d_{ji}/b_{Cl2}) + pt) + (1 - x) \cdot 2 \cdot (l_{Cl2} + (d_{ji}/b_{Cl2}) + pt). \tag{6.8}$$

This equation includes bot the latency and the bandwidth parameters of the network of cluster $Cl2$ as well as the message size d_{ji} and a computation setup time pt. Altogether, the additional overhead is not a constant factor, it increases with larger message sizes, i.e. with larger I/O requests. The I/O time required at the client cluster $Cl1$ and the intercluster network I remains the same, since the additional overhead occurs only at $Cl2$. Additionally to this increased I/O message exchange, two servers are required to fulfill a request. Those two servers are blocked during the computation and delay concurrent requests of other clients. In the optimal case, where data is not transferred between servers, only one server is blocked.

So for each message that needs to be transferred to another server than the originally contacted one, the I/O time increases. As the client-server mapping in MEMFS currently is solely determined by a round-robin assignment, the data matching of client requests can reach an arbitrary small value, up to zero, when only data of other servers than the assigned one is accessed. This shows that optimization of the MEMFS client-server mapping is necessary to improve I/O performance.

6.2 Potential Optimization Strategies for Client-Server Mapping

This section discusses the client-server mapping approaches of the parallel file systems introduced in section 2.2. Following this, other potential improvements to the mapping of clients to servers are presented, in particular an approach that predicts data accesses before they are actually performed and another approach that utilizes hints passed to the file system to describe upcoming data requests.

6.2.1 Request-Based Client-Server Mapping

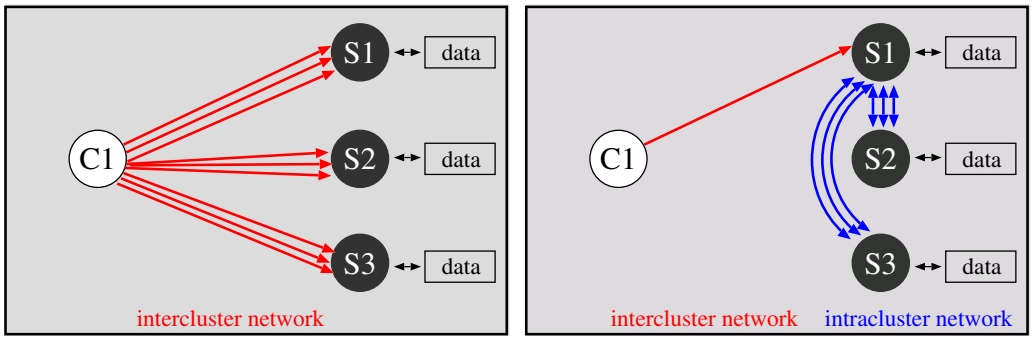
In most intelligent-server parallel file systems (see section 2.2.2) each client has the information about which server(s) to contact for a request and directly communicates with that server(s). We call this *request-based client-server mapping* because the appropriate servers are determined for each single client request and no permanent assignments exist. The client splits all requests into parts for specific servers and transfers each part directly to or from the server that stores this part of the file. This means that the original I/O request can result in multiple requests of the client to different servers.

PVFS2 is a file system that utilizes this technique. Data in PVFS2 is stored at designated I/O servers, the so called *pvfs2-servers* (see [28]). PVFS2 does not map clients to specific servers, the clients contact servers based on particular requests. Each client contacts any one of the pvfs2-servers at startup. This servers replies with configuration information about the file system, including the partitioning of the handle space into ranges for specific I/O servers [28]. Using this handle partitioning a client can compute the responsible server for a given filename. When first accessing a file the client contacts this server and receives a file data distribution function. With this distribution function the client can compute the parts of the file that each server holds.

Each I/O data request can then directly be sent to the server that holds the requested data or is designated to store the current data by evaluating the distribution function. This distribution function can not be changed during runtime, so the cached version of the client does not need to be updated.

Lustre [30] uses a request-based server access, too. The concept is similar to the PVFS2 approach, so the single steps are not described again.

There is a clear advantage of a request-based client-server mapping compared to static mappings where clients contact one specific server until the mapping is explicitly changed. In the request-based approach the clients read and write data directly at the servers that hold the data. Requests of data distributed among multiple servers are split into several client requests, one for each server involved. This means that no server-to-server exchange of I/O data is necessary, because all data is directly transferred from the client to the designated server. This reduces the number of hops on server side to 1. Given the cost model introduced in section 5 this means that $h_{Cl_2} = 1$, so all factors that include the number of hops are excluded



(a) Request split on client side (b) Request split on server side

Figure 6.2: Comparison of requests split at client side or at server side

from formula 5.2, resulting in:

$$a_{Cl2} = l_{Cl2} + d_{ji}/b_{Cl2} + pt. \tag{6.9}$$

This term equals the optimal case in the round-robin mapping as defined in equation 6.7. With this approach the data matching defined in section 6 for a request r and given servers S is at its' maximum because all requested data is located at the contacted server: Let there be a set

$$req = \{r_1, r_2, \ldots, r_n\}$$

with

$$\bigcup_{i=1}^{n} r_i = r.$$

Then the data matching function for each element of this set returns 1:

$$g(r_i, s_j) = 1, \ r_i \in req, s_j \in S, i = 1, \ldots, n.$$

The splitting of r into req and the corresponding mapping of each r_i to a server s_j is done by the algorithm that splits one request into multiple requests for specific servers, for example by using the distribution function in PVFS2.

To achieve sequential consistency between requests of different clients the servers still need to communicate, but they do not need to exchange the usually much larger I/O data. The problem of guaranteeing sequential consistency in a parallel file system is introduced in section 3.3.4.

One drawback of the request-based client-server mapping is that when requests are split into subrequests for specific servers, potentially many small requests are sent out by an I/O client. This means that the number of messages traversing the network

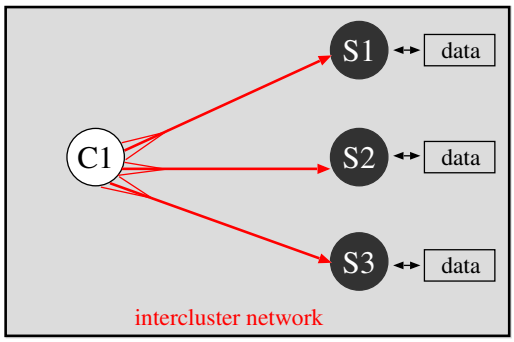


Figure 6.3: Request split and combined at client side

between clients and servers can significantly increase. Since clients and servers in most cases are located on different clusters this means that more messages are sent through the intercluster network interconnect. Inserted into our cost model this yields that the usually high intercluster network latency l_I can become a critical part for the overall I/O performance. See figure 6.2 for a comparison between a request split on client side (a) and a request split on server side (b). This figure disregards that also in case 6.2(b) potentially multiple messages are transferred over the intercluster network interconnect. That happens when the request size exceeds the maximum network message size (see section 5). Anyhow the relatively large maximum network message size prevents this message split from becoming a bottleneck since a request is only split into few, still comparably large messages. But in the case of requests explicitely split at client side, the number of messages can substantially increase, especially when clients use small chunk sizes as necessary for applications with multidimensional data partitioning and nested-strided accesses (see section 4.2). A way to avoid this split into multiple messages is to distribute file data accordingly to the data access schemes of the I/O clients. This approach is described in section 7.2.2.

The scenario displayed in 6.2(a) can be improved by merging all requests for specific servers into one request again on client side (see figure 6.3). This merge of requests requires that new datatypes are automatically computed for each involved server to transfer the potentially complex I/O data scheme in a single I/O operation, i.e. the file view needs to be split into separate file views for each I/O server.

6.2.2 Direct Block Access

Clients of applications that use shared storage file systems (section 2.2.1) either directly access blocks of storage devices or forward requests to remote servers that

then perform blockwise access to attached storage. A GPFS client knows which storage device to use for writing and reading data by information contained in the metadata distributed over the file system [27]. So there is no mapping of a client to a fileserver, but the data is directly accessed. This concept is very similar to the request-based mapping, because requests are split into parts for specific servers on client-side. The difference is that shared storage file systems access data blockwise and intelligent servers file systems use file descriptors (section 2.2). The direct block access always transfers complete file blocks [8], resulting in increased network traffic especially for strided patterns that only access small parts of blocks.

6.2.3 Predicting Accesses

Many processors [55, 56], hard disks and operating systems are loading data into a cache before it is actually accessed. This is a technique known as *prefetching* [57, 58, 59] or *prepaging* [60].

The client-server mapping tries to optimize the mapping of clients to access the servers that hold the requested data. For basic clients that do not know which server to access for each request, the servers have to determine a mapping. One way of optimizing this mapping is to predict future client access parameters on the server side. Prefetching also has to decide which file data to prefetch, comparable to the prediction of parameters. Therefore, the techniques applied in prefetching algorithms can be partially utilized to predict the next accesses.

As described in section 2.2.1, GPFS is a file system that uses prefetching to read data into the buffer pool before it is actually accessed. It recognizes sequential and different strided access patterns [24]. Since many parallel applications show a high regularity in the access of files (see section 4.2) and use recurring stride sizes it is a promising approach to utilize an algorithm that automatically recognizes several sequential and strided access patterns. It can be used to predict recurring accesses with a constant stride factor as in figure 6.4 for example. When the last n (a reasonable value for n has to be found) accesses all use a constant stride it is likely that request number $n+1$ also uses this stride size. The algorithm can then predict this stride for the next access. This approach has the advantage that no changes to the application are required, the prediction is automatically done based on recent accesses. For applications with recurring access patterns this can be very effective, it becomes difficult or impossible the less regular the applications' data access is.

Furthermore n already completed I/O operations are required to predict the next

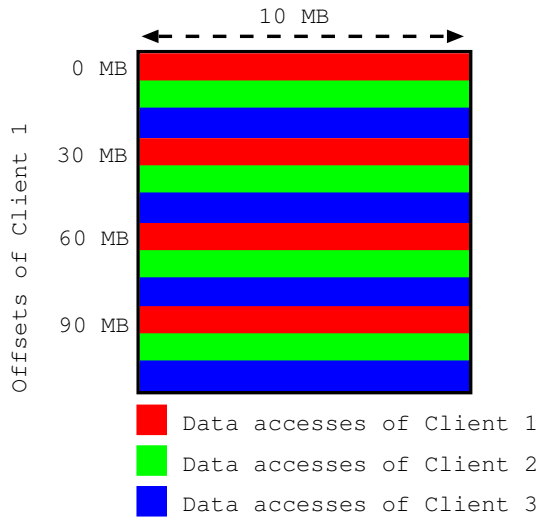


Figure 6.4: Simple strided access pattern for 3 clients. Offsets of clients increase at 30 MB per request.

access, so the first n accesses can not take advantage of the prediction algorithm. An option is to store the access patterns of previous runs in a configuration file for every application. If an application runs many times and every client always works on approximately the same data, this information can be used to predict accesses. Then also the first n accesses can be predicted because information from the previous runs is available. This approach becomes difficult when the number of clients changes between each application run, because then the accessed data for each client usually changes, too.

The prediction of upcoming I/O requests also needs to take knowledge about access patterns into account. The file views that are usually set for each application process define the accessible file regions of each client and can therefore be seen as hints to the file system (see section 2.1.2). The parallel file system needs to filter out all data regions that can not be accessed by a client and restrict the prediction to the accessible regions. After the file view is initially set for each client the algorithm assigns the server that holds the first data entries of the individual file view. The algorithm stores information about previous request sizes and access patterns and uses these information to predict the next access. After a file view is changed, the prediction has to be reevaluated. The algorithm assigns the server that holds most of the data that is predicted to be accessed in the upcoming I/O request for that client. This is a speculative approach, where upcoming requests are predicted based on data gathered at previous requests. If the data access patterns of an application are very irregular, this approach can not help to improve the I/O performance. The prediction of accesses tries to optimize the client-server mapping to reach the optimum data

matching for each access. As introduced in section 6, multiple servers can hold parts of the accessed data. Since this approach does not split requests at client side, data needs to be transferred between servers for all accesses with a data matching lower than 1. As discussed in section 6, the server-to-server transfer increases the number of hops, resulting in increased I/O transfer costs as introduced in equation 6.8.

6.2.4 Mapping based on Hints

GPFS supports a way to prefetch data of other access patterns than the ones natively supported by passing these custom patterns to GPFS as hints via an interface [24]. GPFS then prefetches data for upcoming requests according to these passed hints. This enables GPFS to prefetch data of applications with irregular access patterns, that can not be automatically detected. It requires that the programmer has knowledge of the applications' access patterns and implements the hint passing.

Another work on prefetching data by passing hints to the file system is described in [59]. The authors describe a parallel I/O system where multiple parallel hard disks share a common buffer used for caching and prefetching of data. There, upcoming data requests are passed to an algorithm, which then computes an optimized allocation of the shared buffer. This algorithms in detail can not be used to optimize the client-server mapping of a parallel file system, because it optimizes the usage of a shared buffer in terms of minimizing parallel I/O requests to disk and not the assignment of clients to servers. But the general approach to pass some (or potentially all) of the upcoming data requests out of the application to an algorithm which computes optimal parameters for an underlying system is also interesting for a parallel file system. Using this information about upcoming requests, the file system can compute an optimized client-server mapping.

The hints that should be passed to the file system are:

- Upcoming request offset
- Upcoming request size

Besides these hints, the file system also needs to consider the file view set for each client, as described in the previous section. Accesses of clients are always relative to the file view and therefore the hints must be combined with the information contained in the file view. Based on the given hints and the file views, the algorithm can now compute, which I/O server stores most of the data of the next client request, i.e. maximize the data matching of the next request. The client-server mapping can

then be updated according to this information. This procedure needs to be executed before each I/O request, requiring an additional MPI-IO operation called by the client, the MPI_File_set_info operation which passes hints to the file system. The two hints described above need to be passed to the file system using this function. The new mapping will then be returned to the clients as a parameter of the server-response to this MPI-IO operation. The increased work required on the application source code is the main drawback of the hint-based mapping. The developer needs to compute the parameters of each upcoming I/O request and pass them to the file system. So each I/O request requires one additional server-client communication (the exchange of the MPI_Info object).

Hint-based mapping as well as mappings that predict accesses do not split requests like the request-based client-server mapping. Therefore, these techniques always try to assign the server that holds most – but not neccessarily all – data of the upcoming request. Both algorithms can not always reach a data matching of $g(r, opt) = 1$ for a given request r, resulting in the same additional costs for I/O data transfers between servers introduced in equation 6.8. The difference between the hint-based mapping and the prediction mapping is the approach to reach the maximum data matching. Both try to assign server *opt* to the client, the prediction-based algorithm by predicting upcoming client requests, the hint-based algorithm by actively passing upcoming request parameters to the servers.

Passing hints to the file system is not speculative in contrast to the predictive approach presented in section 6.2.3, as no accesses are predicted, but the information is actively given by the application.

6.3 Usable Strategies in a Parallel Memory File System

This section discusses the usability of the introduced strategies in the parallel memory file system MEMFS.

The direct block access (section 6.2.2) as implemented by most shared storage parallel file systems is not suitable for MEMFS. There, data is either directly accessed via a SAN or through servers that provide blockwise access to locally attached storage. MEMFS, however, is accessed by TUNNELFS through an ADIO device, where data is accessed on a file-basis. Furthermore, a memory file system does not store data on a SAN.

The request-based client-server mapping (section 6.2.1) of most intelligent server parallel file systems like PVFS2 or Lustre would also require some basic changes to the TUNNELFS-MEMFS-cooperation. The currently used interface described in section 6.1.1 allows to change a client-server mapping after specific MPI-IO operations. In the request-based approach, however, the clients themselve have to decide which server is contacted for a specific request or, more precisely, for a part of a request. The mapping has to be actively set on the client-side. The client has to separate a request into parts for specific servers and directly communicate with these servers. MEMFS, however, is only active on the server side and relies on TUNNELFS to tunnel MPI-IO requests to the I/O servers. Enabling the clients to contact the right I/O servers for each request therefore requires to adjust the TUNNELFS client-server communication. This would soften the separation of the two ADIO devices TUNNELFS and MEMFS, complicating the independent development of both. Currently, TUNNELFS provides a simple interface to MEMFS to change the client-server mapping. This interface is well-defined and results in minimal dependency of both devices. Introducing MEMFS-specific parts on the TUNNELFS client side would mean that changes on one of both devices would directly affect the other one, requiring much higher coordination in the development of both. So in general, request-based client-server mapping can be introduced into MEMFS, but the addressed impact on the TUNNELFS-MEMFS architecture has to be considered when choosing one of the presented approaches.

The predictive client-server mapping (section 6.2.3) based on file views fits the architecture of MEMFS and TUNNELFS well. The two devices are developed for MPI-IO operations, where the usage of file views is very common. The file views set by clients are already passed to each server by TUNNELFS. This is especially helpful when reassigning a client to a new server, because all servers (including the newly assigned one) already have knowledge about the file view of that client and are automatically updated when this file view changes. The consistency of these file views is managed in the TUNNELFS layer, so no additional overhead needs to be introduced to distribute file views between servers. The analysis of the file views and the former I/O requests can be done on the server side, since all required information is available there. Then, the existing interface can be used to pass changes in the mapping to the clients. This does not require any changes on the client (TUNNELFS) side and does not weaken the separation of TUNNELFS and MEMFS. Furthermore, the client application does not need to pass any additional information to the servers, because the prediction is done based on previous I/O requests.

The same applies for the mapping of clients to servers based on information given by

the application through passing hints (section 6.2.4). All hints set in the application are already automatically passed by TUNNELFS to the destination file system as MPI_Info objects. MEMFS can evaluate this information and assign a new server based on it through the TUNNELFS-MEMFS interface described in section 6.1.1.

Both the predictive and the hint-based technique are limiting the assignment of new servers to clients by the number of read and write requests a client makes. The design of TUNNELFS requires to complete the active MPI-IO operation before a new assignment can be made. MEMFS can only change the client-server mapping through the provided interface once an I/O request is complete (compare to section 6.1.1). This stands in contrast to the method of assigning a file view and accessing all file data required by a client with a single MPI-IO call, as recommended in section 2.1.3. Applications using this technique can not take advantage of a client-server mapping implementation that uses the existing interface between TUNNELFS and MEMFS (predictive or hint-based client-server mapping). So these two client-server mapping approaches can only optimize the I/O performance of applications that perform multiple I/O requests comparable to the standard Unix-approach.

6.3.1 Choice of a client-server mapping approach

Optimizing the I/O performance of a parallel file system requires a sophisticated combination of both client-server mapping and data distribution techniques. Therefore, the next chapter describes optimizations for the distribution of data between I/O servers. Only after that, combinations of both techniques will be compared and one will be chosen for implementation in the parallel memory file system MEMFS. The client-server mapping only describes the coupling of I/O clients and I/O servers. All of the strategies presented in this chapter try to assign a client to the server that holds most of the requested data. But without knowledge about how data is distributed between the servers, the client-server mapping potentially reach arbitrary poor data matchings. Furthermore, higher data matching values can be reached by explicitly storing data according to the access scheme of clients. This means only a sophisticated combination of both client-server mapping and data distribution algorithms can optimize the I/O performance of a parallel file system. Two potentially good standalone solutions that are contrary to each other can not reach the target performance.

The overall problem can be seen from two perspectives. On the one hand the client should contact the server that holds most of its accessed data. This is done in the client-server mapping. On the other hand the clients' file data should be

distributed to the server that the client accesses. The data distribution algorithms try to optimize this part. These two perspectives do not stand in contrast to each other, but they need to cooperate with each other. Therefore, the next chapter presents data distribution algorithms and following this, combinations of both are analyzed.

Chapter 7

Data Distribution Schemes

The second optimization approach of this thesis is the distribution of file data between I/O servers. This chapter first presents the current state in the parallel memory file system MEMFS and then introduces potential data distribution optimizations that can help to improve the I/O performance of parallel applications.

Distribution of file data is the partitioning of a file F on a subset of the available I/O servers S. The distribution function defines a partition element p_i for each server $s_i \in S = \{s_1, s_2, \ldots, s_l\}$, which means that server s_i stores all data defined in p_i. The first requirement for a partitioning pattern is that the partition elements of all servers define the complete file F:

$$F = \bigcup_{i=1}^{l} p_i. \tag{7.1}$$

This requirement guarantees that each byte of the file maps onto *at least* one partition element (compare with [7]). The data distribution strategies presented in this chapter use different partitioning patterns to distribute file data among some or all of the I/O servers S. We only consider distribution strategies that use non-replicated data, so the second requirement for a partitioning pattern is that the partition elements describe non-overlapping file regions:

$$p_i \bigcap p_j = \emptyset, \forall i, j = 1, \ldots, l, i \neq j \tag{7.2}$$

This second requirement insures that each byte maps onto *at most* one partition element. So together, both requirements define a mapping for each file byte onto exactly one partition element, i.e. exactly one server [7]. The mapping defines a bijective function between the file and the partition elements.

The most important part of a distribution algorithm is the utilized partitioning pattern. The presented strategies of this chapter differ in the utilized partitioning

pattern. Some distribution strategies also redistribute file data to react on changing access patterns of clients.

The main goal of the data distribution is to optimize the applications' I/O performance. Therefore, it needs to

- balance load between servers,
- utilize the multiple paths between clients and servers that a parallel I/O system provides,
- maximize the data matching of client requests and
- optimize for typical access patterns of parallel applications like nested strided data access.

7.1 Current State in MEMFS: Striping

MEMFS currently uses a block-based distribution scheme to partition files between servers. Files are stored in *server blocks* and these blocks are distributed among the servers, always beginning at the server with the lowest rank in the server communicator. This mechanism is know as *striping* and is supported by most parallel file systems, including PVFS, PVFS2, GPFS, Lustre and GFS. The size of the server blocks is called the *stripe size*. Striping is the standard approach to utilize the multiple paths to storage devices in parallel file systems. It balances load between the I/O servers and utilizes the multiple paths of a parallel I/O system by placing parts of a file on each server. It does not maximize the data matching and is also not optimized for typical parallel access patterns like nested-strided data partitioning.

In MEMFS, the stripe-size bs can be set by the application for each file by passing a hint to the file system when creating the file. If it is not passed, a standard value is chosen. This makes the computation of the servers holding some part of the file easy for contiguous data requests. Given an offset off, the corresponding first data block is computed with:

$$b_1 = \frac{off}{bs}. \tag{7.3}$$

Together with the total number of I/O servers $|S|$ the server $s \in S$ that holds the data block $b1$ can be computed with

$$s = \lfloor b_1 \rfloor \bmod |S|. \tag{7.4}$$

With a request size rs the total number of accessed blocks is

$$nb = \lceil \frac{rs}{bs} \rceil + \lfloor \frac{(rs \bmod bs) + (off \bmod bs)}{bs} \rfloor. \tag{7.5}$$

The first summand divides the request size by the blocksize and rounds the result up to consider cases where either the first or the last block is accessed partly, i.e. when the client request either starts or ends at an offset that is not a multiple of the blocksize. The second summand is necessary to regard the special case where both the first and the last block are accessed partly. In this case, one additional block needs to be added to the overall sum, which is done by adding up the two carries, dividing it by the block size and then rounding this result down. Altogether, this request accesses nb blocks that are blockwise distributed among the I/O servers, beginning at server s. The server originally contacted by the client can exchange data with all servers that hold blocks of this request using the formulas introduced above. This means the server that receives the original request can easily decide which server(s) to contact to fulfill the client request.

There are several drawbacks to the current MEMFS approach. The first one results from the distribution always beginning at the server with the lowest rank. This means that the servers with the lower ranks potentially store more data than the ones with the higher ranks and are therefore more frequently accessed by clients, resulting in load imbalance. This effect is especially present when the chosen stripe size is relatively large and the application opens multiple small files. The problem is illustrated in figure 7.1, where the blocks of file 1 are distributed among all servers, but dependent of the size the files 2 and 3 are only placed on a subset of all servers always starting at server rank 0. There are several ways to improve this approach, that are described in section 7.2.1.

The second drawback arises from the fact, that MEMFS currently always transfers the whole data of an I/O request to or from one I/O server. Since striping distributes file data among multiple servers, client requests can involve multiple servers. This is especially true for large requests, that exceed the MEMFS stripe size bs. Data then needs to be transferred between the servers, the data matching introduced in section 6 is less than 1. For a request of size rs byte, at least

$$max(0, rs - \lceil \frac{nb}{|S|} \rceil \cdot bs) \tag{7.6}$$

byte need to be transferred between the servers. The second subtrahend subtracts the maximum amount of file data stored at a single server from the overall request size, giving the minimum amount of client data that needs to be transferred to

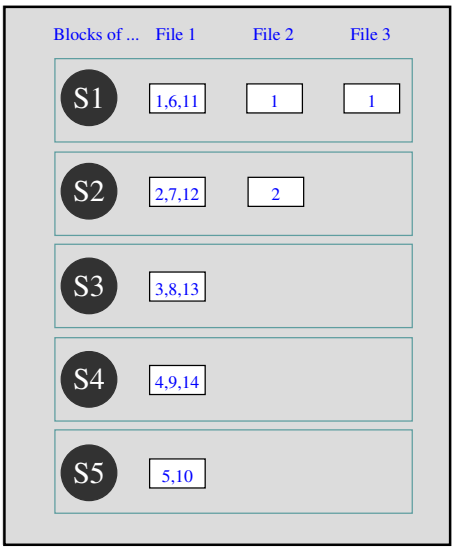


Figure 7.1: Data Distribution of Multiple Files

another destination server. This second drawback can be solved by using the request-based client-server mapping presented in section 6.2.1.

7.1.1 Choice of an optimal stripe size

Many parallel file systems that stripe I/O data between servers allow the user to set the stripe size according to the requirements of the application. PVFS2 uses a standard stripe size of 64 Kbyte [36], while the standard value of GPFS is 256 Kbyte [24] and both allow the setting of custom values.

Given the stripe size *bs*, the I/O server storing the logical file data stream changes every *bs* byte. Most I/O and network devices deliver best results for the transfer of large blocks, because of relatively high latency, which in general counts for large stripe sizes. However, large stripe sizes can also negatively influence the I/O performance. Parallel file systems are utilized to balance I/O load between multiple servers and to utilize the multiple available connections to these servers (see chapter 2). A large stripe size prevents a balanced distribution of data to servers, so that file blocks are potentially concentrated on a subset of the available I/O servers, a problem already illustrated in figure 7.1. File data can only be evenly distributed to servers if the file size is a multiple of the stripe size. The stripe size that delivers the best I/O performance is directly related to the access scheme of an application, the general implication is that large files and large request sizes argue for a large stripe size and vice-versa.

7.2 Potential Optimization Strategies

This section first describes a potential enhancement of the current block-based data distribution of the parallel memory file system MEMFS. Following this, an approach to distribute file data according to the access patterns of the applications' clients is introduced which is for example utilized in the parallel file system Clusterfile [18].

7.2.1 Intelligent Block-Based Distribution

As already described in the previous section, the striping currently used in the parallel memory file system MEMFS especially suffers from load imbalance problems. One way to avoid this is to improve it by several "intelligent" parameters, that help to optimize the distribution of file data between the I/O servers. The following three striping parameters can be tweaked:

1. The beginning of a distribution: The distribution of file data does not always have to start at the server with the lowest rank.

2. The participating servers in a distribution: Specific files could only be distributed to a subset of the available I/O servers.

3. The stripe size: The server block size could be adjusted according to the requirements of a specific application.

We call this improvement of a block-based round-robin distribution an *intelligent block-based distribution*. The three parameters identified above need to be set to best fit the demands of each specific application. Therefore, the algorithm can take several information into account:

- The overall file data stored at each I/O server: To balance load, servers should store similar amounts of data.

- The available memory of each server: In heterogeneous environments some servers can store more data than others.

- The load of each server: Highly loaded servers should be disburdened.

- The expected file size: The chosen blocksize should correspond to the expected file size (see section 7.1.1).

- The number of participating I/O clients: More clients also require more paths to data for optimal I/O performance, so the number of participating servers can be increased, while a file that is only acessed by a single client also only needs to be stored on a single server.

- The access patterns of clients connected to each server (file views, request sizes, strides between requests and other parameters): The placement of data on the I/O servers needs to be coordinated with the mapping of clients to servers. By intelligently setting the three parameters "beginning of a distribution", "participating servers" and "stripe size" according to expected or known client access patterns, the data distribution can yield higher data matching.

Some of these information are available to the file system, like the server load parameters. Others are speculative or need to be given to the file system as hints. For example the file system itself has usually no information about how much data will be stored by the application in specific files, unless the application explicitly sets the file size before writing the file data. If the application provides these missing information as hints, the file system can adjust to them and optimize the parameters 1 - 3 to best fulfill the demands of the application. The application programmer has to pass these hints, otherwise the distribution algorithm is restricted to the internally available parameters.

The combination and evaluation of the given information should result in optimized values for the three parameters of the intelligent distribution. Some information, however, can potentially contradict each other. Consider for example an environment, where one server is highly loaded, but still has more available memory than the other servers. The high load accounts to exclude this server from a distribution, while the available memory accounts the opposite. The algorithm anyhow has to decide for one specific value for each parameter. The given information can be prioritized and then combined to a determined value, e.g. include or exclude this specific server from a distribution.

To give an example of potential improvements that these intelligent parameters result in, one can think of a large file that will be accessed in large chunks by many clients. This file can be distributed to all servers to reach the maximum available parallelism. Furthermore, the block size can be adjusted to the chunk size of the client requests, so that complete blocks are read and written in single requests and no false sharing occurs. In another example, a medium-sized file that will only be accessed by one single client can be distributed to just one server, potentially the server with most memory available or the server with the lowest load. Furthermore,

the client should be assigned to this server to reduce forwarding costs, requiring a combination with a client-server mapping technique. There are several other special situations that this intelligent block based distribution can handle better than the simple striping. Some of them can only be identified if the application provides special information as hints. But also without these hints, the file system can optimize the distribution based on the available information.

We illustrate the potential optimizations in figure 7.2, a modification of the distribution shown in figure 7.1. Now, the blocksize of file 2 was decreased to distribute the file among all servers and file 3 was completely placed at server 5 (the server that holds one block less of file 1 than all other servers). This is just an example for the intelligent optimizations that the algorithm could make, it is not guaranteed that these new distributions are optimal.

The described improvements only change the server at which the distribution starts, the participating servers of a distribution and the used block size. Therefore, the computations already described in section 7.1 can be reused with only slight modifications. When exchanging the data of this request with other servers, the original server needs to consider that potentially only a subset $s^i \subseteq S$ of servers is included in the storage of this file. Formula 7.3 is extended to include the file-specific block size bs_i for file i in the computation of the start block of a given request with offset off:

$$b_1 = \frac{off}{bs_i}. \tag{7.7}$$

Formula 7.4 now includes the start server $s^{i^{start}} \in S$ of a distribution to determine the start server of a request:

$$s = \lfloor b_1 \rfloor \bmod (|S| + s^{i^{start}}). \tag{7.8}$$

Formula 7.5 is now also extended to include the file-specific block size bs_i:

$$nb = \lceil \frac{r}{bs_i} \rceil + \lfloor \frac{(r \bmod bs_i) + (off \bmod bs_i)}{bs_i} \rfloor. \tag{7.9}$$

The three parameters $bs_i, s^{i^{start}}$ and s^i have to be distributed among all I/O servers, so that each server can compute the target servers for a given request. This is necessary to guarantee that each server can handle requests of any client. These parameters need to be kept consistent among all servers, otherwise I/O operations can show undefined behavior.

This data distribution approach is static, just as the simple striping described in the previous section. The three parameters have to be set before the first write or read

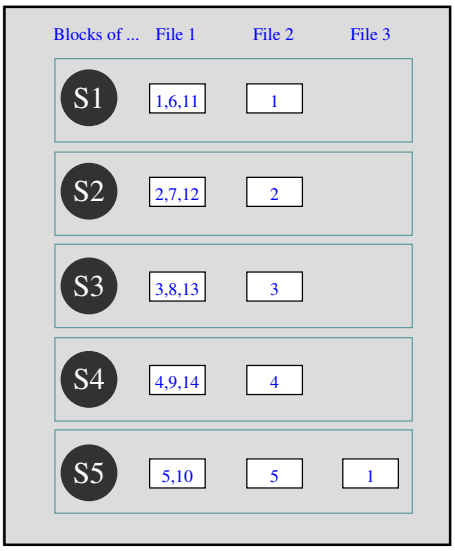


Figure 7.2: Intelligent Data Distribution of Multiple Files

operation occurs. This is necessary to avoid data redistribution and inconsistency of the stored data with the newly set parameters. The setting of these parameters is implementation-specific and can be either done centrally by one designated server (for example the server with the lowest rank in the communicator) or by a distributed algorithm involving all servers.

7.2.2 Partitioning based on Logical Distribution

This section presents a data distribution optimization that physically distributes data according to the logical distribution between application clients. A comparable approach of this optimization strategy is used in the parallel file system Clusterfile [18].

The approach distinguishes between *logical* and *physical* data distribution, where logical describes the distribution of data between the application processes and physical defines the distribution of a file on the server storage devices. The standard physical distribution of parallel file systems is striping file data between multiple servers, as introduced in the last two sections. It can result in poor performance, when an application uses more complex logical distribution patterns that include strides in accesses. According to [18], striping negatively affects the performance and scalability in several ways:

- Striping can result in data fragmentation on the I/O servers and requires complex index computations for each single I/O access, because mismatch of the

accessed data and the distributed data requires complex mapping functions between the logical and the physical distribution.

- Complex access schemes of applications (for example with multidimensional array partitioning) result in the exchange of many small messages, because small parts of data are required from multiple servers. Network devices, however, provide best performance at the transfer of large messages. This increases the network and the server load.

- Contention of accesses at I/O servers can decrease the achievable parallelism between servers. When data is not distributed according to the access schemes of clients, the probability increases that multiple clients access a single server concurrently.

- The mismatch between physical and logical data distribution means poor spatial locality on the servers. The result is non-sequential access of data on the I/O servers, whereas most I/O devices are optimized for sequential accesses.

- The mismatch also increases the probability of false sharing of file blocks between clients.

All these aspects show that striping is non-optimal for complex access schemes that are very common in parallel scientific applications. Those applications can benefit from flexible data distribution. Many scientific applications use multidimensional matrix partitioning between processes, which results in nested strided access patterns (see section 4.2). A sophisticated data distribution should be optimized for this partitioning, but should also support other distribution patterns.

A file system which implements this data distribution algorithm requires knowledge about the logical distribution of the file between the client processes. Usually, this knowledge comes from the file views (section 2.1.2) that are set for each client process. Using this information, the physical distribution can then be related to the logical distribution.

The data distribution algorithm in Clusterfile completely separates the physical from the logical distribution. The logical distribution can differ from the physical distribution to support overlapping file views. In this case, mapping functions are computed between the physical and the logical distribution, which are used to transfer data between files and buffers at read and write operations. This approach also provides flexible adjustment to file view changes. If an application accesses a file with multiple file views, a physical distribution can be chosen that best fits the demands of

the application without requiring data redistribution after each file view change. A basic version of this data distribution strategy defines the physical distribution as the identity of the logical distribution. The file data is then stored according to this file view. In this simplified version, overlapping file views are not supported, because otherwise single logical elements would map onto multiple physical elements, a violation of the requirements introduced in section 7.

A file system can implement its' own partitioning system - as for example Clusterfile with nested PITFALLS [39, 18], or use MPI datatypes for partitioning file data. A file system that uses custom datatypes needs to convert between MPI datatypes and the custom datatypes when using MPI-IO operations. This conversion creates additional overhead. Clusterfile uses its' own partitioning system anyhow, because all mapping and redistribution algorithms are based on the nested PITFALLS [18]. Clusterfile traverses the MPI datatype tree to map it onto a Clusterfile datatype [61].

According to [9], it is expected that file views will be immediately set after a file is opened. An implemented version of this data distribution algorithm should ensure that this behavior is efficient, but also provide the potential to redistribute file data at later application stages, for example after an initial header is written to the file.

In the basic version of this distribution strategy, the file data is directly stored equivalent to the client file views. The data of each client is stored in a *subfile*. If the number of clients is a multiple of the number of servers, each server stores the same amount of client subfiles. If this is not the case, the algorithm stripes the remaining subfiles on disjunct server sets. Given $|C|$ clients and $|S|$ servers, $\lfloor \frac{|C|}{|S|} \rfloor$ subfiles can be stored on single servers. The remaining $|C| mod |S|$ subfiles are striped across disjunct sets of servers to balance the load and the storage amount of servers. The clients whose file views are stored on single server are assigned to the particular server, requiring a combination with the client-server mapping strategy. All accesses of these clients have a data matching of 1 because the complete file region defined in the view of the client is stored at the contacted server. The remaining clients still reach high data matchings, because their potentially noncontiguous file views are striped contiguously on multiple servers. Only these clients require client-server mapping during the application runtime, since data is accessed from different servers.

7.3 Usable Strategies in a Parallel Memory File System

This section describes the usability of the presented data distribution algorithms in the parallel memory file system MEMFS.

The intelligent block-based data distribution is an enhancement of the currently used distribution algorithm. The intelligent parameters can be easily integrated into the current system. Most of the currently used computations only require minor changes. The client information required to set the parameters can also be easily passed to MEMFS, using the MPI hint mechanism already supported by TUNNELFS and MEMFS.

The data distribution based on the logical distribution of file data requires more changes to the existing MEMFS approach. Files have to be stored independently of the sequential file stream. MEMFS can use *virtual* files that explicitly store the file parts of one client. The whole file can then be rebuild as the union of all virtual files. Furthermore, request parameters (offsets and request sizes) need to be set according to the logical distribution of a file. Currently, these logical information are mapped onto the sequential byte stream. This distribution approach also requires coordination with the utilized client-server mapping algorithm, to set the file server of a client to that server that is designated to store the clients data. The adjustments do not change the overall design of MEMFS and only require modification of specific parts of the data distribution parts of MEMFS.

Concluding this chapter, we see that both presented data distribution optimizations can be utilized in the parallel memory file system MEMFS.

Chapter 8

Combination of Client-Server Mapping and Data Distribution Techniques

The last two chapters presented different approaches for the mapping of clients to servers and for data distribution between servers. This chapter now discusses the advantages and disadvantages of combinations of both subtopics. Based on this evaluation, one combination will be chosen for implementation that best fits the demands of the parallel memory file system MEMFS. The goal is to show performance improvements made by client-server mapping and data distribution techniques compared to the original unoptimized version of MEMFS.

In chapter 6, the following approaches M were presented for the mapping of clients to servers:

- *M1*) Round-robin client-server mapping (section 6.1),
- *M2*) Direct data access (section 6.2.2),
- *M3*) Request-based access (section 6.2.1),
- *M4*) Prediction of accesses (section 6.2.3) and
- *M5*) Mapping based on hints (section 6.2.4).

Chapter 7 then introduced the following data distribution techniques D:

- *D1*) Block-based round-robin distribution (section 7.1),

- *D2*) Intelligent block-based distribution (section 7.2.1) and
- *D3*) Distribution of data based on logical partitioning (section 7.2.2).

A sophisticated combination of solutions for both subproblems can optimize the communication between clients and servers in I/O requests. First, it can help to increase the data matching of client requests by storing data at designated servers and assigning clients to these servers. Second, it can minimize reconfiguration of client-server mappings by evaluating application access patterns and storing data according to them.

The following discussion will only include reasonable combinations of elements in M and D. The round-robin based client-server mapping approach described in *M1* does not need to be considered any further, as this is a very limited technique that is already implemented in the current version of the parallel memory file system MEMFS.

This thesis focuses on parallel file systems that operate in the intelligent server architecture, described in section 2.2.2. These intelligent server file systems operate in a client-server architecture, where I/O requests are made on a file basis. Furthermore, in these systems the clients are not connected to direct attached storage like SAN. *M2*, in contrast, is an approach for shared storage architectures (section 2.2.1) with block-based access to direct attached storage devices. Therefore, this technique will also not be considered any further.

The technique *D1* is a special case of *D2*, where all variable factors are set to constant values: The start server of a distribution is always the server with the lowest rank in the communicator, the file is always distributed among all servers and the server block size is always set to either a standard value or – if provided – to a value given by the application as a hint. Therefore, the basic approach with constant values is not treated as a separate technique, instead it is considered as a special case of approach *D2*.

8.1 Potential Combinations

Altogether there remain three approaches for the client-server mapping, namely *M3*, *M4* and *M5* and two approaches for data distribution, *D2* and *D3*. As all remaining elements of M can be combined with all elements of D there exist six potential combinations. These will now be analyzed according to several criteria:

- The first criterion is the expected performance of the combined techniques. In this thesis, the I/O performance of scientific applications should be optimized which states the expected performance of a newly introduced strategy as the most important factor. Evaluating the performance, however, is not easy. Some techniques are expected to show enormous performance improvements for specific application types, while not being usable in other application areas. Therefore, this evaluation will concentrate on the access patterns of typical parallel applications, as described in chapter 4.

- Second, the approaches will be compared with respect to implementation complexity. This master thesis is limited in time, so the chosen techniques need to be realizable in this time frame.

- Third, the usability in MEMFS is discussed. This criterion does not evaluate the overall quality of the discussed approaches, but since one of the presented combinations should be implemented in MEMFS, this is still an important factor for further consideration of an approach in this thesis.

- Furthermore, other special aspects of each combination are discussed, that have either positive or negative influence on the choice of a technique.

The six combinations are now first evaluated for themselves. These evaluations will summarize the discussions of the previous two chapters. After this, the combinations are briefly compared to each other and one is chosen for implementation in MEMFS. The discussion will focus on the usability in an abstract standard parallel file system, to compare the quality of the combinations in general. A restriction is that all techniques are only evaluated in terms of usability in intelligent server parallel file systems, which is done to focus the discussion on one special architecture. This is especially required for comparability of the presented approaches, since shared storage parallel file systems impose completely different requirements on the integrated client-server mapping and data distribution approaches. Only after this, the special suitability in the selected parallel file system MEMFS will be evaluated.

8.1.1 Request-Based Mapping and Intelligent Block-Based Distribution (M3 and D2)

The request-based client-server mapping is expected to clearly improve the I/O performance compared to the standard approach of round-robin client-server mapping. Requests are split on client side and directly transferred to the target servers, so no

overhead on server-side is created to exchange parts of a file stored at other servers. Section 6.2.1 already mentioned that a potential performance bottleneck can arise, when the split of the original client request generates numerous client-server data transfers. This section also introduced a potential solution to this bottleneck by merging all requests for a specific server. The intelligent block-based distribution is not optimized for typical access patterns of parallel applications. It suffers from the problems introduced in section 7.2.2. It is expected to show best performance when accessing large contiguous blocks, because data is distributed between the I/O servers in this manner. Accessing the data in that way optimally utilizes the I/O and network devices, which are optimized for transfer of large contiguous blocks of data.

For applications that access data in large blocks, this combination of techniques seems to be optimal in terms of performance, because only low overhead is generated and large messages can be exchanged between a client and its target servers. However, more complex access patterns like the very common nested strided accesses are generating a significantly larger overhead, because in the block-based distribution the physical distribution of file data is independent of the logical distribution. The nested strided patterns access many, small parts of a file separated by holes (the stride). The native approach accesses each small part individually, which can easily become a bottleneck, because of high I/O and network latency. The usage of data sieving generates large, contiguous requests again, but requires locking of large file regions and can serialize client requests by that, which can also become a bottleneck. Data sieving also transfers more data over the network than required (compare to section 2.1.5).

As already mentioned, this technique does not match the design of MEMFS and TUNNELFS well, because requests are split on the client side, where MEMFS is not present. This would require changes of TUNNELFS, weakening the separation of the two ADIO devices.

The computation of the servers that are accessed in a client request as described in section 7.2.1 is more complicated than in the simple block-based distribution. Therefore, either the variable parameters *block-size*, *involved servers* and *start server* have to be stored at client side, further introducing MEMFS techniques on TUNNELFS side. Alternatively, this combination of techniques can chose to fallback to the non-intelligent block-based distribution, which does not require these parameters.

The complexity of implementation is on an average level: It would require to implement the splitting of requests on client side. The servers are then either contacted with potentially many small data fragments or these fragments are merged, which

would require dynamic building of server-specific file views for each client. The latter approach is clearly more complex.

The request-based client-server mapping is already introduced in many intelligent server parallel file systems like PVFS2 [28] or Lustre [30, 31], but there it is not combined with an intelligent block-based data distribution scheme. PVFS2 uses it's own distribution mechanism where data is either distributed by striping or based on information given by the user as hints. This is not the same like intelligent distribution, where some parameters are automatically determined by the file system. Therefore, combining request-based mapping and intelligent block-based distribution would be especially interesting in terms of comparing the performance to PVFS2.

8.1.2 Prediction of Accesses and Intelligent Block-Based Distribution (M4 and D2)

One special aspect of this mapping approach is that it is speculative because accesses are predicted. This prediction can only be successful if data accesses are somehow regular. But as already described, scientific applications show a very high regularity in their accesses (section 4.2) and are therefore potentially well-suited for combination with a prediction mechanism that analyzes previous accesses. More generally, applications that use regular access patterns are well-suited for the access-prediction algorithm. There, upcoming request parameters are based on previous requests. Since many parallel scientific applications use very regular accesses this technique is expected to show high success in predicting accesses. The prediction of accesses is a promising approach for this kind of applications, especially when the algorithm is optimized for the very frequent nested strided accesses. As mentioned in section 4.2 less than 10% of all files analyzed in the CHARISMA project were accessed with more than two different intervals (strides) between requests. When an application switches very irregulary between two or more strides the upcoming requests can become unpredictable, but based on the observations regarding regularity in parallel scientific applications it can be expected that these irregular switches are very uncommon. The prediction of upcoming accesses based on recent accesses and given file views should therefore show substantial performance improvements for typical applications. In contrast to the request-based mapping, client requests are completely transferred to / from one server, which requires server-to-server data transfers for many requests. Therefore, the expected performance is lower than that of the request-based mapping, while still accelerating I/O operations.

The usability in MEMFS is given, as mentioned this approach can use the existing interface between TUNNELFS and MEMFS to set new client-server mappings. As described, this interface only allows to pass new mappings when an I/O access is complete. Therefore, applications that access complete files in a single I/O call can not be accelerated with this approach. In contrast, the request-based mapping can also accelerate single I/O calls because these are then directed to the right server(s).

The complexity of this implementation is lower than the previously described approach. The server has to analyze all client accesses and predict future accesses based on these information and the given file views. All required information like block sizes are already available on server side, so these do not need to be additionally passed between clients and servers.

8.1.3 Mapping Based on Hints and Intelligent Block-Based Distribution (M5 and D2)

The hint-based mapping can obviously only optimize the client-server mapping if the hints are actually given by the application. When this is provided by the application and all required hints are correctly passed, the file system can compute the right server(s) for upcoming I/O requests and assign each server to the clients accessing it. Since hint-based mapping is not speculative this will result in better performance than the prediction of upcoming accesses, because no false-prediction is possible. This technique also suffers from the requirement of I/O data transfer on server side, as requests are always completely transferred to and from one specific server.

This combination can easily be utilized in MEMFS. Hints are already passed to MEMFS by TUNNELFS, not requiring any changes to this mechanism. Setting hints is a standard construct of MPI for direct optimization of file system accesses. MEMFS can analyze the information given as hints and use the existing mapping interface to set new client-server mappings. The variable parameters of the intelligent block-based distribution do not need to be transferred to the clients, since all mapping and distribution decisions are made on server side.

The implementation complexity of this approach is very low. The implementation of the hint-based mapping needs to interpret optimization hints that are passed to MEMFS. Based on these information the algorithm has to update the client-server mappings, regarding the parameters of the intelligent block-based distribution.

There is one big drawback of this approach. The hints have to be actively passed to the file system by the application. It requires changes to the applications' source

code and introduces overhead. The parameters of each I/O request need to be passed to the file system, before the request is actually performed, generating additional client-server communication.

Hint passing can definitely be a reasonable approach for some applications, where the source code is given and which are actively optimized for the target file system. It is not possible to include these hints, when the source code of the application is not available.

8.1.4 Request-Based Mapping and Distribution of Data Based on Logical Partitioning (M3 and D3)

Combining these two techniques is not a promising approach, because they address the same problem from two different directions. The request-based mapping assigns clients to the servers that hold the requested data blocks, while the distribution based on logical partitioning stores all data of a client on one specific server. For those clients that store file data at only one server, the request-based mapping will always assign the same server, not requiring any of the sophisticated techniques introduced in this approach. Clients whose file data is distributed between multiple servers can benefit from the request-based mapping, but the distribution information needs to be present at the client (TUNNELFS) side. Since this introduces even more MEMFS parts on TUNNELFS side than in the combination of $M3$ and $D2$, this approach does not seem applicable in this special file system domain.

8.1.5 Prediction of Accesses and Distribution of Data Based on Logical Partitioning (M4 and D3)

The main drawbacks of the previous combinations of client-server mapping algorithms with intelligent block-based data distribution were that they either required basic changes in the design of TUNNELFS and MEMFS (the request-based mapping) or had to transfer parts of the I/O data on the server side again, introducing additional hops (prediction of accesses and mapping based on hints). The combination of a prediction of accesses for client-server mapping and data distribution based on logical partitioning is able to resolve both of these drawbacks. Since the physical distribution of file data on I/O servers is directly related to the logical distribution, no transfer of file data between I/O servers is required for those clients that store the complete file views on a single server. Furthermore, these clients only need to be

mapped to one server for each specific I/O request, not requiring any changes to the interface between TUNNELFS and MEMFS. When client file data is completely stored at one designated server, the mapping used here is not really a prediction of the next access, instead a client is directly assigned to the designated server. The prediction is only used for those clients, whose file data is distributed between multiple servers. As described, this data distribution technique only distributes file data of some clients, if the number of clients accessing a file is not a multiple of the number of I/O servers. A remapping of a client to a server is only necessary when the logical distribution of file data (the file views) change.

Still, there are some drawbacks to this approach. First, changing file views requires complex movement of file data between I/O servers. An application is expected to change its' file view very rarely, which means that these additional costs incur seldom. Second, the basic approach of distributing data according to the logical partitioning is not able to deal with overlapping file views of clients. When the physical partitioning of file data is the identity of the logical distribution, file views are not allowed to overlap. Otherwise, multiple elements of the physical partitioning would map to one file byte, a contradiction to the requirement 7.2 introduced in section 7. Section 7.2.2 also introduced the more general approach of a variable physical distribution, which is for example used in the parallel file system Clusterfile. There, the physical partitioning is not necessarily the identity of the logical partitioning, so overlapping file views do not violate requirement 7.2. However, this general approach is introducing a much higher complexity, requiring complex mapping functions between physical and logical distributions.

Furthermore, this approach can only improve I/O performance, when data is actually partitioned between clients. If this is true and the file views are not frequently changed, this combination is expected to reach very high I/O performance. If an application does not partition file data between clients, it cannot take benefit from the improvements introduced in this technique. In this case, the file system needs to be able to fall back to another approach of distributing file data. As previously described, the request-based mapping provides high I/O performance for standard contiguous I/O requests. So when data is not partitioned between clients with complex nested-strided patterns, using this request-based approach as an alternative seems expedient. The previous section stated that a combination of those two techniques does not seem reasonable, but the potential usage stated here is not a combination of both but a composite approach of dealing with different application access patterns. The two strategies will not be used simultaneously for a single file, but alternatively according to the applications data access.

8.1.6 Mapping Based on Hints and Distribution of Data Based on Logical Partitioning (M5 and D3)

This combination also suffers from the problem that the special features of the client-server mapping are not required with the data distribution based on the logical partitioning of a file. The hint-based mapping gives information about upcoming request to modify the client-server mapping appropriately. But the data distribution algorithm used in this combination stores the file data of most clients on one server. The accesses of these clients do not require the setting of hints about upcoming offsets and request sizes. Again, the clients that stripe their views over multiple servers can benefit from this mapping solution, but in general most hints will be dispensable, generating unnecessary overhead.

8.2 Choice of a Combination

This section summarizes the results of the discussions on combinations of client-server techniques with data distribution schemes. Based on this summary, the most promising combined approach is chosen that is implemented for the parallel memory file system MEMFS. The implementation should demonstrate that the theoretical analyzes of the previous chapters can be proven by results collected in a real parallel file system.

As mentioned in the previous sections, some combinations can already be excluded from this summary. Namely, these were the combination of distribution of file data based on the logical partitioning with either the request-based mapping or the hint-based mapping. As these two combinations are excluded now, only four potential combinations remain.

Table 8.1 summarizes the evaluation of the previous sections and shows the differences of the combinations in direct contrast, regarding the four criteria introduced previously.

The combination *M3* and *D2* promises very good performance for large block requests, but suffers from the mismatch of the logical and the physical distribution of file data when using more complex access patterns. It furthermore has several drawbacks. It does not fit the design of MEMFS and TUNNELFS well and is comparatively complex to implement. Furthermore, the basic approach of directly accessing the target servers by splitting up requests is already extensively used and analyzed in existing parallel file systems like PVFS2 and Lustre. The scientific innovation would

be the combination with the intelligent block-based data distribution. It further would be interesting to analyze its implementation in a parallel remote memory file system to see if remote memory access can provide better results than a local parallel file system.

Combining *M4* and *D2* will result in improved I/O performance, although it is expected to be lower than the performance of the previously mentioned combination. Still, other aspects argue for choosing this prediction algorithm over the request-splitting. It is well usable in the TUNNELFS-MEMFS-architecture, since the existing interface can be reused and all computations can take place on the server side, where MEMFS is present. Furthermore, it is a completely new approach, not used in any existing parallel file system to our knowledge. Therefore, comparing the performance of this new technique to those of existing file systems would deliver some interesting results. It could be a challenging and potentially successful task to tune this algorithm in a remote memory file system to reach the performance of existing parallel file systems with request-based client-server mapping. It could be especially interesting to analyze the performance of different application types to evaluate situations where this approach is or is not well suited.

The combination of *M5* and *D2* promises good I/O performance with a good usability in MEMFS and comparatively low implementation complexity. Unfortunately, it requires application changes (active passing of hints). Since MEMFS should be simply integrated into existing MPI-IO applications the hint-based approach is inapplicable for the general case. It potentially can be used in the future to further improve the client-server mapping for applications that provide these hints.

Combining *M4* and *D3* is expected to result in very good I/O performance for the common nested-strided access patterns of parallel applications. For applications that do not partition file data between clients, a fall-back to either a simple standard approach or to one of the other strategies described in this chapter is required. As the combination of data distribution based on logical partitioning is expected to reach the best I/O performance for typical parallel scientific applications, this solution is chosen to be implemented in MEMFS. As this thesis is limited in time and is mainly interested in the potential reachable I/O performance improvements, we decide to implement the direct mapping of logical distributions to physical partitions. As described, this direct mapping uses the identity of the logical distribution as the physical distribution and therefore does not support overlapping logical distributions. But as a positive effect it can dispense on using complex mapping functions between logical and physical distributions. To become usable in a real production parallel file system, it needs enhancements that support overlapping file views. The

Name	**Performance**	**Usability**	**Complexity**	**Specials**
$M3 + D2$	very good	poor	medium	like PVFS2
$M4 + D2$	good	good	medium	Speculative
$M5 + D2$	good	good	low	Requires appl. changes
$M4 + D3$	very good	very good	medium	Only non-overlapping views

Table 8.1: Comparison of Presented Approaches

implementation of this technique is kept modular to support later add-on of this functionality. For applications that do not partition file data we decide to reuse the existing standard MEMFS approach, which already showed good I/O performance for simple I/O patterns as published in [49].

Chapter 9

Implementation and Evaluation

9.1 Implementation Issues

This section discusses the required changes to MEMFS and TUNNELFS to implement the prediction-based client-server mapping and the data distribution based on the logical distribution of file data. These changes are not described in detail as the specific implementation adjustments are not relevant for the purpose of this thesis, but special considerations regarding the overall design of a parallel file system are explained.

9.1.1 Client-Server Mapping

To support dynamic mapping of clients to servers, a file system must distribute file views of clients between servers. A client usually sets a file view at one specific server. As long as the client only contacts this server for I/O requests relative to this file view, this is not problematic. But as soon as the mapping assigns another server to the client, the new server also requires the file view information. A file system can choose to always distribute file views between all servers or to distribute file views only on request. TUNNELFS always distributes newly set file views between all I/O servers. This mechanism is not cost-intensive since views can be represented compactly. When a file view is set by a client, the receiving server needs to distribute the following information to all other servers:

- file id: The unique identificator of the file for which the file view is set.
- client rank: The rank of the client in the global communicator to identify the client associated with this file view.

- displacement, etype and filetype: The three parameters that define the file view, as described in section 2.1.2.

When combining a client-server mapping with the data distribution based on logical distribution of file data, the mapping does not change for the clients that store file data at only one server after it is initially set. Still, the file data of some clients is distributed among multiple servers (compare to section 7.2.2). The data defined by the views of all clients with ranks lower than $\lfloor \frac{|C|}{|S|} \rfloor \cdot |S|$ is distributed round-robin-wise to one specific server, the file data of the remaining clients is striped across all servers. The client-server mapping retrieves previously stored information about the number of clients which opened a file to compute those clients that potentially require mapping updates. These clients require a dynamic mapping to servers, where distributed file views are required. After specific I/O operations (namely open, set_view, read and write operations) a MEMFS function is called, which updates the client-server mappings. For those clients that store file data on a single server, the mapping is retained unchanged, for the other clients the prediction-based algorithm computes a new server. The updated mappings are then passed via TUNNELFS to the clients, which contact the newly assigned server in the next MPI-IO operation.

9.1.2 Data Distribution

The distribution function of MEMFS was changed from the standard block based striping to a distribution which is the identity of the file view of a client. In this first implemented version the distribution stores all views of clients in separate sequential *subfiles*. This means that a subfile stores all data defined in the view of a client. These subfiles can either be completely stored at one server or striped across multiple servers. They are stored on single servers when the number of subfiles is a multiple of the number of servers, so that each server then stores the same amount of subfiles. Otherwise some or all subfiles are striped across multiple servers. MEMFS currently does not evaluate the amount of described data in each view. Clients can define different amounts of data in file views, so that an even distribution of clients to servers does not always balance the stored data amount. In future versions, information retrieved from the file view description can be used to balance the amount of data distributed to each server.

Each subfile is internally stored in a separate file in MEMFS. The filename is the original filename with the client rank appended as a suffix. Since MEMFS files are only accessed through the MPI-IO interface, no problems with other interfaces that

list directory contents can occur.

This simple MEMFS model does not support overlapping file views. The subfiles are the identity of the file views and as defined in section 7 each file byte must map onto exactly one partition element. To support overlapping file views, a different layer that maps potentially overlapping file views to non-overlapping subfiles would be necessary. In MEMFS, however, this simple distribution is utilized to evaluate the achievable performance increase with physically distributed data according the logical distribution.

Data redistribution is currently also not supported. A workaround is to manually read all file data, open a new file with new file views and write all data again with these new file views. A simple way to automatically support data redistribution could be to do these steps internally in MEMFS, hidden to the user. A more sophisticated data redistribution also requires mapping functions between separate data distributions.

The MEMFS approach should be a simple approach to show potential improvements when data is distributed according to file views.

Since only applications that partition file data among processes can utilize the new distribution algorithm, this needs to be activated by a file hint. This hint can be given for any file separately, so that applications that process multiple files can store non-partitioned files with the former distribution algorithm and partitioned files with the new distribution based on the logical partitioning.

9.2 Experimental Results

To evaluate the performance achieved by the new data distribution algorithm, two different I/O benchmarks were performed.

The first benchmark performs different MPI-IO operations to write and read contiguous parts of files. In this benchmark, all processes access a shared file. The user can pass the file size fs as a parameter to this benchmark, which allows measurements with different amounts of data. The client process with rank $c_i \in C = \{c_1, c_2, ..., c_k\}$ then accesses the contiguous file region reaching from offset

$$\frac{fs}{|C|} \cdot (i-1)$$

to offset

$$(\frac{fs}{|C|} \cdot i) - 1.$$

Each process writes its amount of data in this region, then reads the data back and verifies the correctness of the read buffer. The results of this benchmark are presented in section 9.2.1 with a comparison to the same benchmark ran with PVFS2.

The second application is a development of the Indiana University, which uses MPI-IO operations to distribute a matrix between multiple processes [62]. Minor changes were made to this application to benchmark the performance of the MPI-IO calls. The application defines a 3-dimensional matrix which is distributed blockwise among the participating processes. The blockwise matrix distribution is defined as a MPI datatype (see section 2.1.1) with the MPI_Type_create_darray constructor. See the MPI standard [63] for the parameters of this constructor. This newly defined datatype is then used as the fileview for the MPI-IO data transfer. Similar to the first benchmark, each application process writes its data amount, reads it back and verifies it for correctness. The results of this benchmark are discussed in section 9.2.2.

All measurements were performed five times to exclude negative influence of other running processes, high network load, etc. The results presented in the next two sections always illustrate the best performance of these five runs.

Both benchmarks were performed on the WR cluster, a 6-node cluster, where each compute node is equipped with 4 AMD Opteron 846 64-bit processors at 2 GHz, 8 GB PC3200 main memory and one 36 GB SCSI disk with 10.000 rpm. The compute nodes internally communicate over a Myrinet PCI-X Host Interface M3F-PCIXD-2. For a more detailed description of the cluster hardware setup see [64]. The runs that included a second cluster were additionally started on the PCC cluster, a 32-node cluster. Each of these compute nodes is equipped with 2 Intel Xeon processors at 2.66 GHz and 1 GB main memory. The internal communication is performed over 1 Gbit/s Fast Ethernet. The two clusters are physically separated with a distance of around 20 kilometers and are connected through the VIOLA network [40], which provides dedicated 10 GBit/s optical WAN connections. Each of the 6 WR cluster nodes and the 32 PCC cluster nodes is equipped with one Fast Ethernet network adapter of 1 Gbit/s connected to the VIOLA network, resulting in a maximum transfer rate between the two clusters of 6 Gbit/s.

9.2.1 Contiguous Data Transfers

This benchmark compares the performance of the data distribution based on logical distribution implemented in MEMFS to results collected with PVFS2. The benchmark

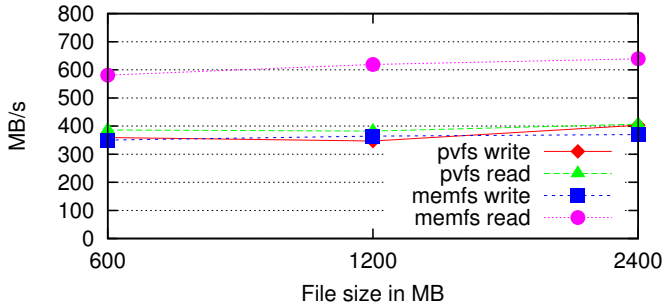


Figure 9.1: Contiguous benchmark including PVFS2 and MEMFS measurements

performs non-overlapping, contiguous data accesses and is therefore usable in the newly implemented data distribution approach, although it cannot leverage the real advantage of this distribution. Data is not accessed in strided patterns, which means that the file views already define contiguous parts of the file. The main advantage of a data distribution based on logical partitioning is that it performs better than standard striping with strided access patterns (compare to section 7.2.2). Anyhow, this benchmark should show that this approach can also be used effectively for contiguous non-overlapping I/O requests. Two different setups where chosen to compare the performance to the results of an existing, widely used parallel file system, PVFS2 in this case. The measurements were taken with varying file sizes from 600 MB to 2.4 GB as illustrated in figure 9.1.

The first setup measures the performance of PVFS2 by placing one PVFS2 server on each of the 6 WR cluster nodes. This was empirically found to deliver the best results, placing more than one PVFS2 server delivered worse performance, probably because of the single shared disk at each cluster node. The read and write operations perform very similar in PVFS2, both reach a maximum value of about 400 MB/s (see *pvfs* results of figure 9.1). Each of the SCSI disks can perform at a maximum transfer rate of about 70 MB/s, so PVFS2 is able to approximately reach the overall maximum of $6 \cdot 70$ MB/s $= 420$ MB/s. The file system overhead is compensated by the caching of file data.

The second setup measures MEMFS with the data distribution based on the logical distribution, in this case a contiguous, non-overlapping distribution. In this setup the WR cluster was coupled with the PCC cluster to demonstrate the usage of remote main memory. As only 12 nodes of the PCC cluster were available for reservation and the other nodes running a long-term production job, 12 servers were placed on the WR cluster and 12 clients processes on the PCC cluster. In MEMFS, multiple servers started on each cluster node can provide higher performance than single servers started on each node. In the ccNUMA architecture of the WR cluster, the

different processors of each node can access the main memory concurrently, without the serialization that occurs when accessing a shared disk. In this setup, MEMFS is able to reach more than 600 MB/s read performance to remote main memory (see *memfs read* in figure 9.1). The write performance is below this value, at a level comparable to PVFS2 of about 400 MB/s. We assume that this lower write performance compared to the read performance partially results from the required allocation of memory in the write case. In [49] we already demonstrated that MEMFS is able to nearly saturate the available 6 GigE connections of the WR cluster with MPI-IO read operations and a total of 18 servers. Taken togehter, these results show that a parallel file system for main memory that stores data on a remote cluster over a high-performance network can exceed the results of a standard disk-based parallel file system like PVFS2.

9.2.2 Strided Data Transfers

This benchmark demonstrates the improvements of the new distribution algorithm when using strided data accesses. As mentioned in section 4.2 these strided accesses are very common in parallel scientific applications and a parallel file system used in these environments should optimize for them. The 3-dimensional distribution of the 3-dimensional matrix in this application results in complex, nested strided access patterns (compare to figure 4.1). The file data distribution based on the logical distribution stores each clients data according to this access scheme. Therefore, no complicated mechanism like data sieving (see section 2.1.5) is required to write and read the data efficiently. Unfortunately, no comparisons to PVFS2 were possible, since the data verification part of this application always reported incorrect results for PVFS2. We suspect that these incorrect results are related to the missing support of sequential consistency in PVFS2 (see section 2.1.4), although the developers state that PVFS2 guarantees atomicity of writes to non-overlapping noncontiguous regions (compare to section 2.2.2). These problems need to be examined more thoroughly in the future. Instead, we compared the results of the new data distribution with the former MEMFS approach of striping as described in section 7.1. Figure 9.2 shows the results for 6 servers placed at the WR cluster and 6 clients placed at the PCC cluster. Figure 9.3 illustrates the same measurements taken with 12 servers on the WR cluster and 12 clients on the PCC cluster, respectively. In both figures "block" denotes the results of the striping distribution (block-based) and "view" denotes the data distribution based on the logical distribution (with file views).

The application distributes a regular 3-dimensional matrix blockwise between mul-

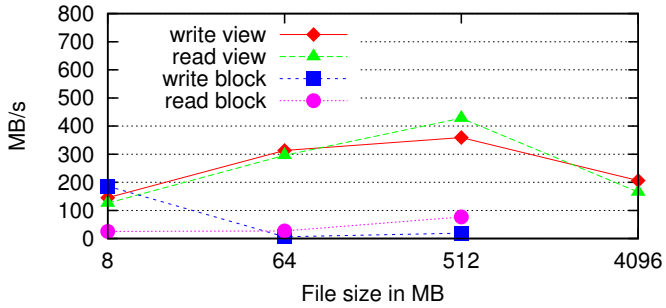


Figure 9.2: Strided benchmark for MEMFS with two different data distribution schemes, 6 servers and 6 clients

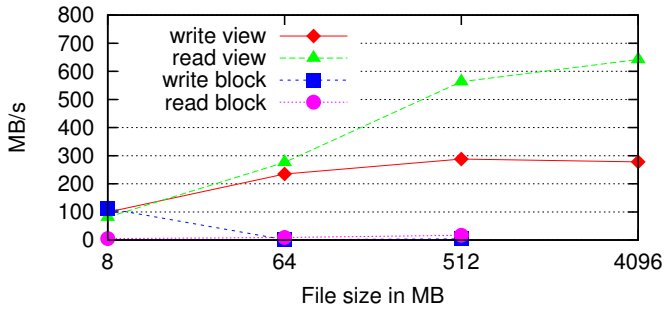


Figure 9.3: Strided benchmark for MEMFS with two different data distribution schemes, 12 servers and 12 clients

tiple processes. We varied the *size* parameter of this application to measure the performance of different file size. Since the application builds a 3-dimensional matrix of integers and in our system integers are 4-byte datatypes, the resulting file size is computed as $size^3 \cdot 4$.

The figures 9.2 and 9.3 show a very large gap between the block-distributed (striped) results and the view-distributed results for all file sizes larger than 8 MB. The locking mechanism of MEMFS described in section 3.3.4 had to be turned on for the block-distributed version, because it uses the data sieving mechanism of ROMIO (compare to section 2.1.5). The locking mechanism serializes write accesses to overlapping regions, resulting in very poor write performance. But also the read performance of all block-distribution measurements lies below 100 MB/s. These results show the problems of a mismatch between the logical and the physical file layout. With the standard striping each client needs to access data parts from all servers.

In contrast, the view-distribution can reach very good results comparable to the contiguous accesses of the benchmark introduced in the previous section. The best transfer rates are achieved with 12 clients and servers as illustrated in figure 9.3. With a file size of 4096 MB the read operations reach more than 600 MB/s, while

the write operations still reach approximately 290 MB/s. We were unable to run the block-distributed measurements with a file size of 4096 MB because of program errors. We assume that these errors result from the numerous strided accesses generated with this data distribution.

Chapter 10

Conclusion

This thesis presented optimization strategies for parallel file systems. It concentrated on the mapping of clients to servers and the data distribution among the servers. For both topics we presented a basic standard approach used in the parallel memory file system MEMFS and potential optimized techniques. We further discussed combinations of client-server mapping and data distribution techniques and evaluated those combinations against each other with special respect to implementation in MEMFS. In this chapter we now summarize the work of this thesis. We outline the most important results and mention potential advances of the development in the future work section.

10.1 Summary

In this thesis we first introduced parallel I/O systems in general and parallel file system in particular. Different file system architectures were presented with some of todays most widely used parallel file systems. We also presented our previous work on the parallel memory file system MEMFS, which works together with TUNNELFS. To further motivate the work on optimization techniques for parallel file systems we presented results of access pattern analyses that stated concrete requirements for parallel file system. We also introduced a simplified cost model for the I/O time of applications, which can be used to compare different optimization strategies.

One of the two main parts of this thesis is the mapping of clients to servers. We presented the approach of MEMFS, a round-robin mapping of clients to servers, and the technique of most existing parallel file systems: the direct access of requested data at the designated servers. Two other techniques were also discussed, a prediction

algorithm and a mapping based on application hints.

The second main part discussed in this thesis is the distribution of file data among I/O servers. In this part we presented the standard approach of most parallel file systems including MEMFS, the blockwise striping of file data among the server processes. Since the basic version of this distribution had some fundamental drawbacks, this chapter also presented an improved version of striping that utilized more information to determine the distribution parameters. Still, we emphasized some major problems of the general blockwise data distribution approach. It results in numerous small I/O requests and network messages in the very common case of noncontiguous data access. We introduced another data distribution approach that is based on a very common technique, the setting of file views.

We analyzed all potential combinations of mapping and data distribution strategies. For the special demands of the parallel memory file system MEMFS we chose the prediction mapping approach and the data distribution that rebuilds the clients file views. In the results chapter we showed that we are able to deliver high performance with a parallel file system that stores data in the main memory of remote cluster nodes. Our setup showed that access to remote main memory can achieve higher transfer rates than a common disk-based parallel file system installed on a single cluster, PVFS2 in this case. The newly introduced optimization strategies especially showed significant performance increases with noncontiguous access patterns.

The main goal of this thesis was to show that the optimization strategies presented throughout the thesis can help to improve the I/O performance of typical parallel applications in a real parallel file system. We were able to show this performance increase especially with strided data accesses that are very common in parallel applications.

10.2 Future Work

The data distribution algorithm implemented in MEMFS is a basic version of the data distribution based on the logical partitioning of file data. Currently, it neither supports overlapping file views nor changes to file views. To provide support for a wide range of parallel applications in the future, this algorithm needs to be enhanced. Introducing mappings between logical and physical distributions helps to support both overlapping file views as well as file view changes. The mapping functions that need to be used then introduce additional overhead. In the future, we will have to show that the I/O performance of a distribution including these mappings still

exceeds the standard striping performance.

Furterhmore, the extended data distribution and client-server mapping approaches of MEMFS need to be compared to other file systems in more detail. Since Clusterfile uses a very similar data distribution approach, it provides the potential to compare the results of this data distribution strategy in a disk-based file system to a remote memory file system. In these more elaborate comparisons more application types can be evaluated.

Nomenclature

Terms

t	Total I/O time
$Cl = \{Cl1, Cl2\}$	Clusters 1 and 2
$l_{Cli} \in \mathbb{R}_+, i = 1, 2$	Network latency for cluster 1 and 2
$b_{Cli} \in \mathbb{R}_+, i = 1, 2$	Network bandwidth for cluster 1 and 2
$h_{Cli} \in \mathbb{N}, i = 1, 2$	Number of hops at cluster 1 and 2
I	Intercluster network
$l_I \in \mathbb{R}_+$	Network latency of intercluster network
$b_I \in \mathbb{R}_+$	Network bandwith of intercluster network
$C = \{c_1, c_2, \ldots, c_k\}$	Set of Clients
$S = \{s_1, s_2, \ldots, s_l\}$	Set of Servers
$M_j = (m_{j1}, m_{j2}, \ldots, m_{jn_j})$	Ordered set of messages of client j
n_j	Number of messages of client j
$a_{Cli}, i = 1, 2$	Transfer time of cluster 1 and 2
a_I	Transfer time of intercluster network
t_{ji}	I/O time of message $m_{ji} \in M$
t_j	Total I/O time for client j
$off \in \mathbb{N}$	Offset
$disp \in \mathbb{N}$	Displacement
$etype$	Elementary datatype
$ftype$	Filetype
$view = (disp, etype, ftype)$	File view
$rs \in \mathbb{N}$	Request size in byte
$R = (off, view, rs)$	I/O read or write request
opt	server that optimizes the data matching of a request
F	File
bs	block size (stripe size)

p_i	Partition element of server i
b_1	First data block of a request
nb	Number of data blocks of a request

Functions

$f : C \rightarrow S$	Client-Server mapping function
$g : R \times S \rightarrow \mathbb{R}$	Data matching function

Operators and other symbols

mod	Modulo operator

Bibliography

[1] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Top 500 Supercomputer Sites (2006) http://www.top500.org/.

[2] Meuer, H., Strohmaier, E., Dongarra, J., Simon, H.: Performance Development — Top 500 Supercomputing Sites (2006) http://top500.org/lists/2006/06/performance_development.

[3] Grochowski, E., Halem, R.D.: Technological impact of magnetic hard disk drives on storage systems (2002) http://researchweb.watson.ibm.com/journal/sj/422/grochowski.html.

[4] Gropp, W., Lusk, E., Sterling, T.: Beowulf Cluster Computing with Linux. Volume 2. MIT Press (2003) http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=9947.

[5] May, J.M.: Parallel I/O for High Performance Computing. Academic Press (2001)

[6] Pawlowski, B., Shepler, S., Beame, C., Callaghan, B., Eisler, M., Noveck, D., Robinson, D., Thurlow, R.: The NFS version 4 protocol. Proceedings of the 2nd international system administration and networking conference (SANE2000) (2000) 94

[7] Isaila, F.: Clusterfile Parallel File System. publikation, IPD Tichy, University of Karlsruhe, Germany (2004)

[8] Ross, R., Worringen, J.: EuroPVM/MPI 06, Tutorial 2, High-Performance Parallel I/O. Conference Tutorial, Bonn, Germany (2006)

[9] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI — The Complete Reference: Volume 2, the MPI-2 Extensions. Volume 2. MIT Press (1998) http://mitpress.mit.edu/book-home.tcl?isbn=0262571234.

[10] : IEEE Portable Applications Standards Committee. Technical report (2006) http://www.pasc.org/plato/.

[11] Cheng, A., Folk, M.: HDF5: High performance science data solution for the new millennium. (2000) 149–149

[12] Li, J., Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R.: Parallel netcdf: A scientific highperformance i/o interface (2003)

[13] Thakur, R., Gropp, W., Lusk, E.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory (2004) \tthttp://www-unix.mcs.anl.gov/romio/.

[14] Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Technical report, Mathematics and Computer Science Division - Argonne National Laboratory (1996) http://www-unix.mcs.anl.gov/mpi/mpich/.

[15] : Open MPI: Open Source High Performance Computing (2006) http://www.open-mpi.org/.

[16] Thakur, R., Gropp, W., Lusk, E.: An Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. In: Proceedings of the 6th Symposium on the Frontiers of Massively Parallel Computation. (1996) 180–187

[17] Thakur, R., Gropp, W., Lusk, E.: A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In: Proceedings of SC98: High Performance Networking and Computing. (1998)

[18] Isaila, F., Tichy, W.F.: Clusterfile: a flexible physical layout parallel file system. Concurrency and Computation: Practice and Experience **15** (2003) 653–679

[19] Lamport, L.: How to make a correct multiprocess program execute correctly on a multiprocessor. IEEE Trans. Comput. **46** (1997) 779–782

[20] Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. Volume 1. MIT Press (1999) http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3490.

[21] Thakur, R., Gropp, W., Lusk, E.: Data Sieving and Collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press (1999) 182–189

[22] Jung, N.: Speicher und Massenspeichersysteme (in german) (2006) https://www5.inf.fh-bonn-rhein-sieg.de/intern/Jung/Technische_Informati%k_2/Vorlesung/TI2_Folien-09_N.pdf.

[23] Tate, J., Kelly, J., Rm, P., Stehlik, L.: IBM TotalStorage: SAN Product, Design, and Optimization Guide. Volume 3. International Business Machines Corp. (2004) http://www.redbooks.ibm.com/redbooks/SG246384/wwhelp/wwhimpl/js/html/ww%help.htm.

[24] Schmuck, F., Haskin, R.: GPFS: A shared-disk file system for large computing clusters. In: Proc. of the First Conference on File and Storage Technologies (FAST). (2002) 231–244

[25] Jones, T., Koniges, A., Yates, R.: Performance of the IBM general parallel file system. In: Proc. of the Parallel and Distributed Processing Symposium, 2000. IPDPS. (2002) 673 – 681

[26] IBM Corporation: IBM General Parallel Filesystem (2006) http://www-03.ibm.com/servers/eserver/clusters/software/gpfs.html.

[27] Hochstetler, S., Beringer, B.: Linux Clustering with CSM and GPFS. Volume 2. International Business Machines Corp. (2005)

[28] Drakos, N., Moore, R., Latham, R.: Parallel Virtual File System, Version 2: PVFS2 Guide (2006) http://www.pvfs.org/pvfs2-guide.html.

[29] Carns, P.H., Ligon III, W.B., Ross, R.B., Thakur, R.: PVFS: A parallel file system for linux clusters. In: Proceedings of the 4th Annual Linux Showcase and Conference, Atlanta, GA, USENIX Association (2000) 317–327

[30] : Lustre Whitepaper: A Scalable, High-Performance File System. (2002) http://www.clusterfs.com/.

[31] : Lustre Data Sheet: high-performance storage architecture and scalable cluster file system. (2006)

[32] Red Hat, Inc.: Red Hat Global File System (2006) http://www.redhat.com/software/rha/gfs/.

[33] IBM Corporation: IBM High Performance Computing: An Introduction to GPFS (2006) http://www.ibm.com/systems/clusters/software/whitepapers/gpfs_intro.pdf%.

[34] Hlawatschek, M., Grimme, M.: Data sharing with a gfs storage cluster. Red Hat Magazine (2005) http://www.redhat.com/magazine/006apr05/features/gfs/.

[35] Red Hat, I.: Red Hat GFS 6.1: Administrator's Guide (2005) http://www.redhat.com/docs/manuals/csgfs/browse/rh-gfs-en/.

[36] PVFS: Parallel Virtual Filesystem (2006) http://www.pvfs.org/.

[37] Karp, M.: Lustre high performance file system to support infiniband. Network World (2005) http://www.networkworld.com/newsletters/stor/2005/1121stor1.html.

[38] Duessel, T., Eicker, N., Isaila, F., Lippert, T., Moschny, T., Neff, H., Schilling, K., Tichy, W.F.: Fast parallel i/o on cluster computers. CoRR **cs.DC/0303016** (2003)

[39] Ramaswamy, S., Banerjee, P.: Automatic generation of efficient array redistribution routines for distributed memory multicomputers (1995)

[40] The VIOLA Project Group: Vertically Integrated Optical testbed for Large scale Applications (2005) http://www.viola-testbed.de/.

[41] Pöppe, M., Schuch, S., Bemmerl, T.: A Message Passing Interface Library for Inhomogeneous Coupled Clusters. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003), Workshop on Communication Architecture for Clusters (CAC 2003), Nice, France (2003)

[42] Pöppe, M., Scholtyssik, K., Schuch, S., Worringen, J.: MP-MPICH - User Documentation and Technical Notes (2004)

[43] Berrendorf, R., Hermanns, M.A., Seidel, J.: Remote Parallel I/O in Grid Environments. In: Proc. of the Sixth Conference on Parallel Processing and Applied Mathematics (PPAM). Lecture Notes in Computer Science, Poznan, Poland, Springer (2005)

[44] The DEISA Project Group: Deisa - Grid Architecture (2005) http://www.deisa.org/grid/architecture.php.

[45] Latham, R.: PVFS2 performance study on BGW (2005) http://www-unix.mcs.anl.gov/~robl/bgl/bgw.html.

[46] Hermanns, M.A., Berrendorf, R., Birkner, M., Seidel, J.: Flexible I/O in Reconfigurable Grid Environments. In: Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference. Lecture Notes in Computer Science, Dresden, Germany, Springer (2006)

[47] The DEISA Project Group: Distributed European Infrastructure for Supercomputing Applications (2005) `http://www.deisa.org/`.

[48] : Memory technology evolution: an overview of system memory technologies. Technical report, Hewlett-Packard Development Company, L.P. (2006) `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[49] Seidel, J., Berrendorf, R., Birkner, M., Hermanns, M.A.: High-Bandwith Remote Parallel I/O with the Distributed Memory Filesystem MEMFS. In: Proceedings of PVM/MPI 2006. Lecture Notes in Computer Science, Bonn, Germany, Springer-Verlag Berlin Heidelberg (2006) 222–229

[50] Nieuwejaar, N., Kotz, D., Purakayastha, A., Ellis, C.S., Best, M.: File-access characteristics of parallel scientific workloads. IEEE Transactions on Parallel and Distributed Systems **7** (1996) 1075–1089

[51] Wang, F., Xin, Q., Hong, B., Brandt, S., Miller, E., Long, D., McLarty, T.: File system workload analysis for large scale scientific computing applications. In: Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies. (2004)

[52] Kotz, D., Nieuwejaar, N.: File-system workload on a scientific processor. In: IEEE Parallel and Distributed Technology, IEEE Computer Society Press (1995) 51–60

[53] Purakayastha, A., Ellis, C.S., Kotz, D., Nieuwejaar, N., Best, M.: Characterizing parallel file-access patterns on a large-scale multiprocessor. In: Proceedings of the Ninth International Parallel Processing Symposium, IEEE Computer Society Press (1995) 165–172

[54] Carnes, B.: The i/o stress benchmark codes (2006) `http://www.llnl.gov/asci/purple/benchmarks/limited/ior/`.

[55] Advanced Micro Devices, Inc.: 3DNow! Technology Manual. (2000) `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs%/21928.pdf`.

[56] Intel Corporation: IA-32 Intel Architecture Optimization Reference Manual. (2006) http://download.intel.com/design/Pentium4/manuals/24896613.pdf.

[57] Free Software Foundation, Inc.: Data Prefetch Support. (2006) http://gcc.gnu.org/projects/prefetch.html.

[58] Griffioen, J., Appleton, R.: Reducing file system latency using a predictive approach. In: USENIX Summer. (1994) 197–207

[59] Kallahalla, M., Varman, P.J.: Optimal prefetching and caching for parallel i/o sytems. In: SPAA '01: Proceedings of the thirteenth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, ACM Press (2001) 219–228

[60] Tanenbaum, A.S. In: Moderne Betriebssysteme: Der Working-Set-Algorithmus (in german). Volume 2. Pearson Studium (2002) 242 – 245

[61] Isaila, F., Singh, D., Carretero, J., Garcia, F., Szeder, G., Moschny, T.: Integrating logical and physical file models in the mpi-io implementation for "clusterfile". ccgrid **0** (2006) 462

[62] Meglicki, Z.: High performance data management and processing, darray.c. Technical report, Indiana University (2004) http://beige.ucs.indiana.edu/I590/node102.html.

[63] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI — The Complete Reference: Volume 1, the MPI-2 Core. Volume 2. MIT Press (1998) http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=4579.

[64] Berrendorf, R.: WR Cluster hardware specification (2006) http://wr0.wr.inf.fh-bonn-rhein-sieg.de/wr/hardware/hardware.html.

Appendix A

Source Codes

A.1 List of MEMFS ADIO functions

Source: ad_memfs.h

```
1
    void ADIOI_MEMFS_Open(ADIO_File fd, int *error_code);
    void ADIOI_MEMFS_Close(ADIO_File fd, int *error_code);
    void ADIOI_MEMFS_ReadContig(ADIO_File fd, void *buf, int count,
                                MPI_Datatype datatype, int file_ptr_type,
                                ADIO_Offset offset, ADIO_Status *status,
                                int *error_code);
    void ADIOI_MEMFS_WriteContig(ADIO_File fd, void *buf, int count,
                                 MPI_Datatype datatype, int file_ptr_type,
10                               ADIO_Offset offset, ADIO_Status *status,
                                 int *error_code);
    void ADIOI_MEMFS_IwriteContig(ADIO_File fd, void *buf, int count,
                                  MPI_Datatype datatype, int file_ptr_type,
                                  ADIO_Offset offset, ADIO_Request *request,
                                  int *error_code);
    void ADIOI_MEMFS_IreadContig(ADIO_File fd, void *buf, int count,
                                 MPI_Datatype datatype, int file_ptr_type,
                                 ADIO_Offset offset, ADIO_Request *request,
                                 int *error_code);
20  int ADIOI_MEMFS_ReadDone(ADIO_Request *request, ADIO_Status *status,
                             int *error_code);
    int ADIOI_MEMFS_WriteDone(ADIO_Request *request, ADIO_Status *status,
                              int *error_code);
    void ADIOI_MEMFS_ReadComplete(ADIO_Request *request, ADIO_Status *status,
                                  int *error_code);
    void ADIOI_MEMFS_WriteComplete(ADIO_Request *request, ADIO_Status *status,
```

```
                                int *error_code);
void ADIOI_MEMFS_Fcntl(ADIO_File fd, int flag, ADIO_Fcntl_t *fcntl_struct,
                       int *error_code);
void ADIOI_MEMFS_WriteStrided(ADIO_File fd, void *buf, int count,
                              MPI_Datatype datatype, int file_ptr_type,
                              ADIO_Offset offset, ADIO_Status *status,
                              int *error_code);
void ADIOI_MEMFS_ReadStrided(ADIO_File fd, void *buf, int count,
                             MPI_Datatype datatype, int file_ptr_type,
                             ADIO_Offset offset, ADIO_Status *status,
                             int *error_code);
void ADIOI_MEMFS_WriteStridedColl(ADIO_File fd, void *buf, int count,
                                  MPI_Datatype datatype, int file_ptr_type,
                                  ADIO_Offset offset, ADIO_Status *status,
                                  int *error_code);
void ADIOI_MEMFS_ReadStridedColl(ADIO_File fd, void *buf, int count,
                                 MPI_Datatype datatype, int file_ptr_type,
                                 ADIO_Offset offset, ADIO_Status *status,
                                 int *error_code);
void ADIOI_MEMFS_IreadStrided(ADIO_File fd, void *buf, int count,
                              MPI_Datatype datatype, int file_ptr_type,
                              ADIO_Offset offset, ADIO_Request *request,
                              int *error_code);
void ADIOI_MEMFS_IwriteStrided(ADIO_File fd, void *buf, int count,
                               MPI_Datatype datatype, int file_ptr_type,
                               ADIO_Offset offset, ADIO_Request *request,
                               int *error_code);
void ADIOI_MEMFS_Flush(ADIO_File fd, int *error_code);
void ADIOI_MEMFS_Resize(ADIO_File fd, ADIO_Offset size, int *error_code);
ADIO_Offset ADIOI_MEMFS_SeekIndividual(ADIO_File fd, ADIO_Offset offset,
                                       int whence, int *error_code);
void ADIOI_MEMFS_SetInfo(ADIO_File fd, MPI_Info users_info, int *error_code);
void ADIOI_MEMFS_Get_shared_fp(ADIO_File fd, int size,
                               ADIO_Offset *shared_fp,
                               int *error_code);
void ADIOI_MEMFS_Set_shared_fp(ADIO_File fd, ADIO_Offset offset,
                               int *error_code);
void ADIOI_MEMFS_Delete(char *filename, int *error_code);
```

A.2 MPI_File Struct

Source: adio.h and mpio.h

```
1
    typedef struct ADIOI_FileD {
        int cookie;              /* for error checking */
        FDTYPE fd_sys;              /* system file descriptor */
    #ifdef XFS
        int fd_direct;           /* On XFS, this is used for direct I/O;
                                    fd_sys is used for buffered I/O */
        int direct_read;         /* flag; 1 means use direct read */
        int direct_write;        /* flag; 1 means use direct write  */
10      /* direct I/O attributes */
        unsigned d_mem;          /* data buffer memory alignment */
        unsigned d_miniosz;      /* min xfer size, xfer size multiple,
                                    and file seek offset alignment */
        unsigned d_maxiosz;      /* max xfer size */
    #endif
        ADIO_Offset fp_ind;      /* individual file pointer in MPI-IO (in bytes)*/
        ADIO_Offset fp_sys_posn; /* current location of the system file-pointer
                                    in bytes */
        ADIOI_Fns *fns;          /* struct of I/O functions to use */
20      MPI_Comm comm;           /* communicator indicating who called open */
        MPI_Comm agg_comm;      /* deferred open: aggregators who called open */
        int io_worker;          /* bool: if one proc should do io, is it me? */
        int is_open;            /* deferred open: 0: not open yet 1: is open */
        char *filename;
        int file_system;         /* type of file system */
        int access_mode;         /* Access mode (sequential, append, etc.) */
        ADIO_Offset disp;        /* reqd. for MPI-IO */
        MPI_Datatype etype;      /* reqd. for MPI-IO */
        MPI_Datatype filetype;   /* reqd. for MPI-IO */
30      int etype_size;          /* in bytes */
        ADIOI_Hints *hints;      /* structure containing fs-indep. info values */
        MPI_Info info;

        /* The following support the split collective operations */
        int split_coll_count;    /* count of outstanding split coll. ops. */
        MPI_Status split_status; /* status used for split collectives */
        MPI_Datatype split_datatype; /* datatype used for split collectives */

        /* The following support the shared file operations */
40      char *shared_fp_fname;   /* name of file containing shared file pointer */
        struct ADIOI_FileD *shared_fp_fd;  /* file handle of file
                                              containing shared fp */
        int async_count;         /* count of outstanding nonblocking operations */
        int perm;
        int atomicity;          /* true=atomic, false=nonatomic */
```

```
    int iomode;             /* reqd. to implement Intel PFS modes */
    MPI_Errhandler err_handler;
    void *fs_ptr;            /* file-system specific information */
#ifdef ROMIO_TUNNELFS
    int ind_info_change;    /* independent info change, checked in ad_tunnelfs */
#endif
} ADIOI_FileD;

typedef struct ADIOI_FileD *ADIO_File;

typedef struct ADIOI_FileD *MPI_File;
```