Optimization of non-contiguous MPI-I/O Operations

Bachelor Thesis

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Enno David Zickler
E-Mail-Adresse:	e.zickler@gmail.com
Matrikelnummer:	6250134
Studiengang:	Informatik
Erstgutachter:	Dr. Julian Kunkel
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Dr. Julian Kunkel

Hamburg, den 29.01.2015

Abstract

High performance computing is an essential part for most science departments. The possibilities expand with increasing computing resources. Lately data storage becomes more and more important, but the development of storage devices can not keep up with processing units. Especially data rates and latencies are enhancing slowly, resulting in efficiency becoming an important topic of research. Programs using MPI provide the possibility to get more efficient by using more information about the file system.

In this thesis, advanced algorithms for optimization of non-contiguous MPI-I/O operations are developed by considering well-known system specifications like data rate, latency, or block and stripe alignment, maximum buffer size or the impact of read-ahead-mechanisms. Access patterns combined with these parameters will lead to an adaptive data sieving for non-contiguous I/O operations.The parametrization can be done by machine learning concepts, which will provide the best parameters even for unknown access pattern.

The result is a new library called NCT, which provides a view based access on non-contiguous data at a POSIX level. The access can be optimized by data sieving algorithms whose behavior could easily be modified due to the modular design of NCT. Existing data sieving algorithms were implemented and evaluated with this modular design. Hence, the user is able to create new advanced data sieving algorithms using any parameters he regards useful. The evaluation shows many possibilities for where such an algorithm improves the performance.

Acknowledgements

My warmest gratitude goes
to Dr. Julian Kunkel for enabling, supervising and reviewing this thesis while always giving helpful advise and keeping me motivated,
to Professor Ludwig for enabling this thesis,
to Max for the great companionship while writing our theses,
to Florian for profreading,
to my parents for always supporting me even in my 9th semester.
Thanks to Kira for endless nights of discussion, chearing me up and profreading without losing her temper.

Contents

1. Introduction

This chapter introduces shortly high-performance computing (HPC) in general and the problem of accessing non-contiguous data. Based on that, the goal of this thesis as well as a quick overview of the structure are presented.

1.1. Problem Statement

High Performance Computing became part of more and more research fields, as it makes experiments faster, more cost efficient or is essential to make them at all. This wide use of HPC is mainly possible because nowadays super computers could be build from of the shelf hardware by combining them to a cluster. Due to this fact super computers are affordable and highly scalable. Despite - or perhaps more precisely because of - these systems are still tide to the general development of computer hardware. As the computing power of processors scales on Moore's Law, storage and network technology is leaking behind. In consequence of this development and the growing data amount of computing tasks, data input and output (I/O) became a major bottleneck. Research on optimizing I/O systems is therefore a wide field and leads to different approaches, such as specialized file systems or the MPI-I/O standard. This thesis will inspect the field of non-contiguous I/O patterns and in particular data sieving algorithms.

Non-contiguous data access means, that the desired data parts are interrupted by other data. For example in ares of structs, accessing just some fields of the struct for the hole array leads to a repetitive pattern of small parts of wanted and unwanted data. Accessing the desired data parts individually mostly leads to poor performance. It depends on the size of the data parts. For big data parts individual access is appropriate but for small ones it is not. One method to address this problem is data sieving. The idea is to access not only the desired data, but also the gaps between them to a temporary buffer and then sieve out the unwanted gaps. The most popular implementation of this method is provided by ROMIO and implementation the MPI-I/O standard. As this only allows data sieving by using MPI-I/O, data sieving is not as widely used as it could be.

Additionally, modern cluster often have complex storage systems, whose performance depends on a wide variety of parameters. This makes accessing non-contiguous data in an efficient way difficult, as the best access method depends on the storage systems status. As the research on data sieving is quite small most current used data sieving algorithms are not flexible enough to adopt to this varying status of the storage system.

1.2. Goal of the Thesis

Goal of this thesis is to evaluate the performance of current data sieving algorithms in order to implement a new advanced one. The new data sieving implementation should be flexible enough to adopt the data sieving to the underlying system without forcing users to re-think settings for every access pattern. The parametrization should be possible at run time of the program and not force a particular technical optimization that may be a good choice only on certain systems. Instead users should only provide high-level information about their access pattern and intended use that is automatically translated into technical hints based on the system's hardware characteristics. This opens the possibility to adopt the data sieving to the current state of the cluster system like monitoring the current use of the network or the storage system and dynamically change these parameters.
As the ROMIO implementation is tied to the use of MPI and so limits uses cases for data sieving, the new implementation should be usable more independently. The goal of the thesis can be split into the following sub goals:

1. **Analysis of available data sieving algorithms**
   - Extract access pattern where improvements are necessary and possible
   - Find beneficial effects such as stripe alignment to consider them in advance data sieving methods.
2. **Implementation of a MPI independent data sieving library**
   - Making data sieving usable in more programs by providing a POSIX like interface.
   - Provide a simple solution to integrate the library in MPI-I/O
3. **Design improved data sieving algorithms**
   - The new data sieving algorithms should be more flexible by providing more parameters to adopt their underlying systems.

1.3. Structure of the Thesis

In Chapter 2 topics and background information needed to write this thesis were presented as well as some publications in this field of research in the related work section. Chapter 3 starts with evaluating the design of current data sieving algorithms and then explains the design of the new data sieving library implemented for this thesis. Details on the implementation are given in Chapter 4. The evaluation of the algorithms is performed in Chapter 5. In the end Chapter 6 concludes the thesis and lists potential future work.

First, the naive and existing ROMIO algorithm has been implemented, then these results influenced the development of further algorithms. Many iterations of benchmarking followed by improving the design and algorithms were performed. Therefore the order of the chapters does not actually represent the temporal order the work has been performed.

2. Background & Related Work

This chapter introduces topics and background information needed for writing this thesis. Cluster and data storage systems are introduced in Section 2.1 as they represent the used system. MPI especially MPI-I/O, ROMIO and data sieving are software components related to this thesis and therefore also introduced in Section 2.1. Section 2.2 presents related work which is done in the research field of non-contiguous and parallel I/O.

2.1. Background

Knowing the environment, in which the researched topic is usually applied, is important to accomplish a valuable solution. Therefore, a short overview on modern HPC is given in Section 2.1.1 by explaining the structure and key components of these systems. As data storage is complex nowadays it is introduced in Section 2.1.2 separately.
Just as important as knowing the environment is to know about commonly used available solutions for the research problem. In the case of non-contiguous parallel I/O this includes MPI and MPI-I/O and its widely used implementation ROMIO. Both, MPI and ROMIO, are also introduced Sections 2.1.3 and 2.1.4 in this section as well as the idea of data sieving in Figure 2.5.

2.1.1. HPC

Architecture The norm architecture in modern super computers is to interconnect thousands of computers to build a cluster. This concept is possible due to fast interconnections between the individual computer nodes. Most of these compute nodes are build from off-the-shelf processors as they are much cheaper than developing and producing specialized hardware for super computers, like in the beginning of super computing. [TOP15]

Interconnection A fast network to connect all the computing units is needed to run a cluster. The most used technologies to build such a network are Infinitude and Gigabit Ethernet. Together they are used in over 80%, about 43% Infiniband and 27% Ethernet, of the TOP500 supercomputer in Nov 2014. Both technologies are available with different performances. The slowest but most inexpensive one is 1Gb Ethernet with a transfer rate of 1 Gbit/s. 10 Gb Ethernet is capable of 10 Gbit/s and often transferred over optical wires. Infiniband is more expansive but capable to transfer up to 300 Gbit/s in it the EDR version. Infiniband can be used on copper cables only for distance up to 15m. Longer distances need fiber cables. Not only the transfer rates are an important factor, there is also the latency which is typically correlated to the transfer rates and therefore lower on Infiniband. To get these latencies for a system either the vendors specifications could be

considered or could be measured with a simple ping between nodes. Here must be kept in mind that the ping time is the round-trip time and therefore must be bisected.

Operating System Linux has become the major choice in term of operating systems on cluster [TOP15]. The benefit of having such a generic and customizable operating system while providing a well known system to the users seems to influence this development. As there are many Linux distributions it is possible to use small lightweight distributions on compute nodes and more advanced once on I/O and front end nodes do more complex tasks. As Linux is a UNIX like, mostly POSIX compatible operating system, developing software for HPC should run on top of these standards to provide a good portability between systems. POSIX is a standard for UNIX like operating systems and provides many APIs to make programs portable on different operating system. For this thesis especially the standardized API for file I/O is used.

2.1.2. Data storage

Data Storage has become a complex task. In big systems like super computers it is done in specialized networks. There are all kinds of requirements to these systems, like reliability, availability and performance. Therefore, different specialized standards, technologies and file systems were developed. As they still rely on normal storage devices these are introduced first, followed by some network storage solutions and parallel file systems with a closer view on Lustre.

Storage Devices The most used storage device is still the hard disk drive (HDD) although solid state disk (SSD) are catching up they are still too expensive. They are used for faster caching in modern data storage system but the main characteristic of the under laying HDD is still crucial for modern data storage systems. These characteristics are visible in bandwidth and latency. While the limited bandwidth can be overcome by combining multiple drives in RAIDs the latency can not be coped with that easily. The latency of HDD is mainly caused by the fact that the data is stored on a rotating disks and the need of moving a mechanical arm over the desired data on the disk. This leads to varying latency for accessing data, called seek time, which depend on how far the arm has to move. As these seek time varies between 1 and 15 ms this fact is important to mind for non-contiguous data access. The block oriented data storage should be also minded as it is characteristic for most data storage and is also for SSDs. This means that data is stored in blocks and only can be accessed in whole blocks. For a long time this block size was 512 Bytes but as HDDs became bigger the block size has been increased to 4 KiByte. The new HDDs are still capable of accessing 512 Byte blocks at the interface but internally they access 4KiBytes. SSDs on the other hand still use the 512 Bytes blocks as they do not benefit from the bigger access chunks but have a limited amount of access till they break.

Network Storage The need of accessing huge amounts of data from multiple computers leads to a centralized data storage solution. This is possible with the Network Attached Storage (NAS) or Storage Area Network (SAN) concept. In the NAS concept a server provides the data over the network at a file level to the user. Therefore, the server manages

all the hard drives and file systems. In the SAN concept the storage is provided at a block level. Hence, the SAN interconnects all the storage devices and provides a block access on them. On top of that one or multiple servers and file system can be built to store data. This can be seen in Figure 2.1.
For both systems there is a protocol needed to provide data access for the client over network. The most common protocols are Server Message Block (SMB) and Network File System (NFS). They both provide shared access for multiple user on the data. This data access is mostly independent of the servers underlying storage system. This provides an easy use of shared storage. But as both protocols were not designed with massively parallel application they are not optimized for this. With pNFS these functionality is currently developed but not as common as the use of specialized parallel file system with own protocols .

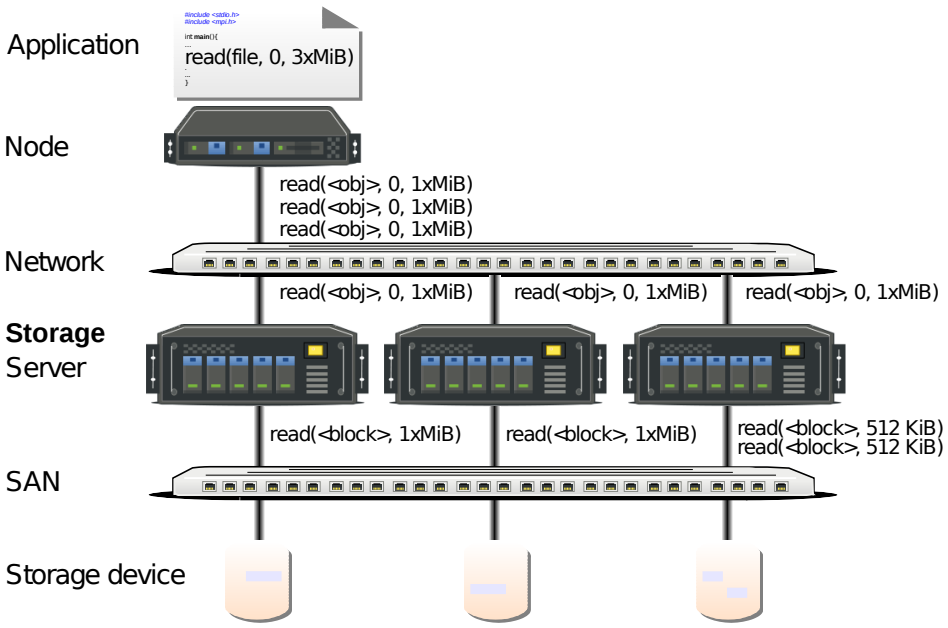

Figure 2.1.: Parallel file system based on SAN [Kun13]

Parallel File Systems Specialized file systems were build to provide concurrent access on data by parallel applications. These can provide better performance for accesses from multiple nodes on a single file. To accomplish this multiple storage server with attached storage are needed. The data is scattered over the storage server so that on the access each server will be used and the performance is accumulated. This concept is implemented by many distributed and parallel file systems such as PVFS, GPFS and Lustre.
A detailed view of Lustre is provided in Figure 2.2. Lustre consists of the following components:

- **OST** The Object Storage Targets, which can be a block storage device from direct attached ones to big SANs.

- **OSS** The Object Storage Servers, which store the data on one or often multiple OSTs.
- **MGS** The Management Server stores information about all Lustre file systems in a cluster.
- **MGT** The Management Target stores the data of the MGS.
- **MDS** The Meta Data Server provides all the meta data like file names, directories or permissions.
- **MDT** The Meta Data Target stores the data of the MDS.
- **Client** The Lustre Client is installed on each compute node and is used to access data from the file system.

In Lustre the data is stored in objects, which are stored on the OSTs. If a client accesses data the meta data server provides the meta data as well as the information in which objects the data is stored. With this information the client can access the file directly over the OSSs on all involved OSTs. Therefore, the performance of all OSTs and the bandwidth of the network between all involved OSSs and the client is used. Lustre uses the follwing terms and definitions for these performance in his manual [Ora11]:

- The **network bandwidth** equals the aggregated bandwidth of the OSSs to the targets.
- The **disk bandwidth** equals the sum of the disk bandwidths of the storage targets (OSTs) up to the limit of the network bandwidth.
- The **aggregate bandwidth** equals the minimum of the disk bandwidth and the network bandwidth.
- The **available file system space** equals the sum of the available space of all the OSTs.

As the OSTs rely on block storage devices the minimum size for an access is 4KiB but Lustre tries to access data in 1MB chunks as this is the size which is transferred over the network in the RPCs. As Lustre tries to optimize the performance an algorithm called read-ahead is implemented. This increases the chunks of each access up to 40MiB if 2 or more contiguous accesses of the RPC size occur. The read-ahead stops immediately if non-contiguous access happens.
Another feature to know of for this thesis is the possibility to set the stripe size and stripe count in Lustre. Stripes are the data portions in which a file is split up to distribute it on the available OSTs in a round robin fashion. There is a default set for the file system, which is usually between 1 - 4 MiB scattered over all OSTs. But it is also possible to set these parameters individually for each file or directory.

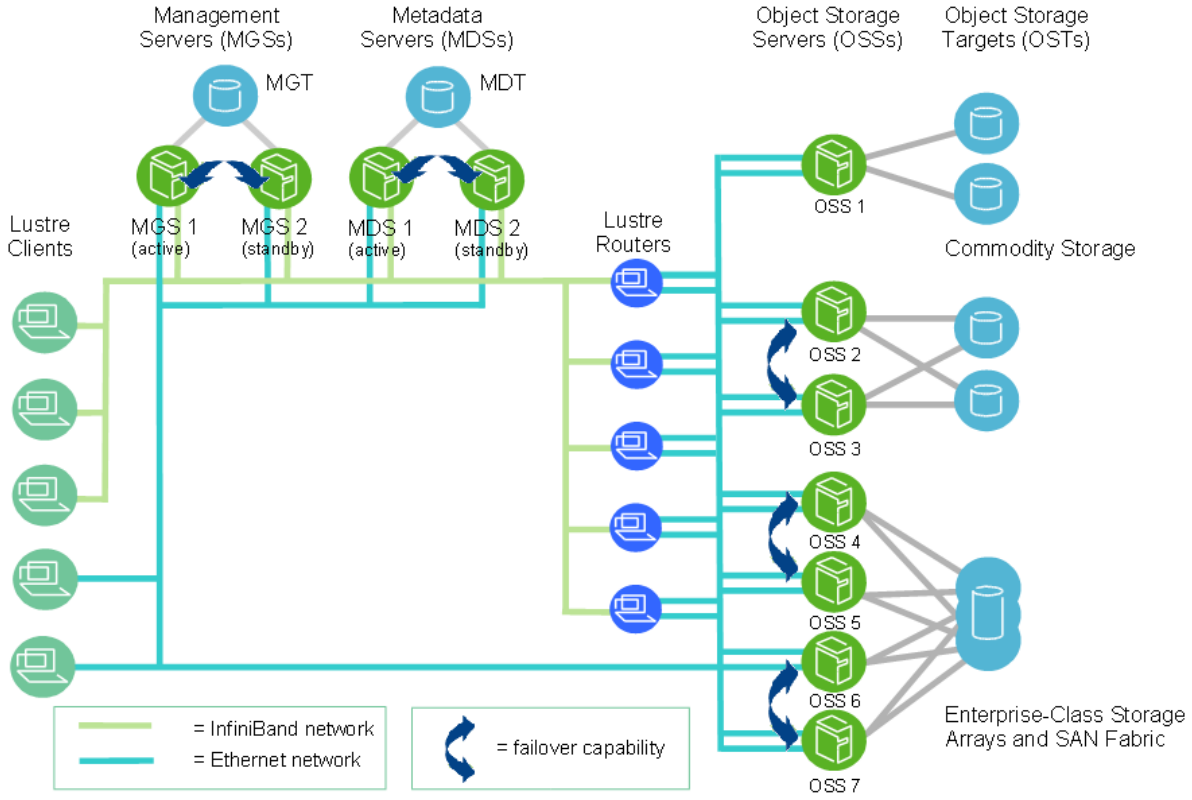

Figure 2.2.: Lustre file system [Ora11]

Caching Caching is a common method to speedup data flow and used in many parts of computers. It improves performance by holding data in nearer and faster memory location than it actually belongs. Therefore its also used by Lustre, operating systems on compute and I/O nodes and the HDDs. Lustre caches data on the client side as well as on the server side, e.g. for the read-ahead. The operating system caches data between system calls from the program and hands the data over to the file system. Modern HDD controller are also caching data. This is always to be kept in mind for performance optimization on data access.

2.1.3. Message Passing Interface

The Message Passing Interface is a standard for portable massage-passing between processes. It is the de facto standard in HPC and therefore implemented in many different versions and running on most super computer. MPI introduces in version 2 a new standard for I/O called MPI-I/O. This allows to implement many optimizations for parallel I/O. One of them is the widely used implementation of MPI-I/O, called ROMIO, which includes the the initial implementation of data sieving by Thakur et. al. [TGL99]. As the ROMIO version is the base where this theses starts with optimization, it is introduced. Before presenting the actual I/O interface MPI defines, the file view is presented.

File and data types in MPI The file view in MPI provides the possibility to give every process a different view of the file. With a set view the process only see the data describe by the view. To create such a view MPIs data types are used. MPI provides all basic

data types as own data types. This is mainly needed as the representation of data types vary on different systems. On heterogeneous clusters, data conversion is needed, by using own data types this conversion can be done by the MPI implementation transparently for the user. To not only provide the basic data types it is possible to create derived data types by using the basic data type. Thus, data structures like structs can be build as MPI data types. For the I/O the possible data conversion of data types is also useful as the data can be stored in a more generic format than the the naive data representation of the system. This provides more probability of the data. In Figure 2.3 a file view build up on one elementary type, called etype, is shown. This etype can be every basic or derived MPI data type.

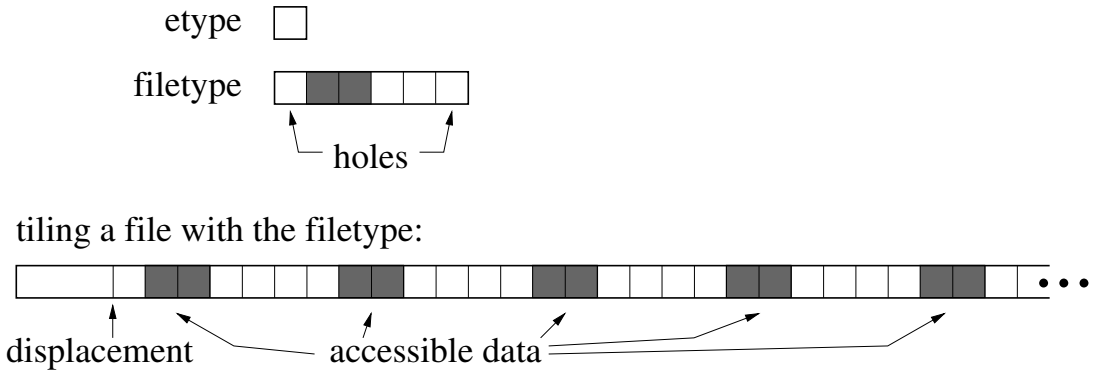

Figure 2.3.: MPI file view [DF12]

MPI-I/O For accessing the data through the file view MPI provides many different functions. They have the orthogonal aspects, positioning (explicit offset vs. implicit file pointer), synchronism (blocking vs. nonblocking and split collective) and coordination (noncollective vs. collective) as described in [DF12] and shown in Table 2.1.

positioning	synchronism	coordination	
		noncollective	collective
explicit of sets	blocking	MPI_FILE_READ_AT MPI_FILE_WRITE_AT	MPI_FILE_READ_AT_ALL MPI_FILE_WRITE_AT_ALL
	nonblocking & split collective	MPI_FILE_IREAD_AT MPI_FILE_IWRITE_AT	MPI_FILE_READ_AT_ALL_BEGIN MPI_FILE_READ_AT_ALL_END MPI_FILE_WRITE_AT_ALL_BEGIN MPI_FILE_WRITE_AT_ALL_END
individual file pointers	blocking	MPI_FILE_READ MPI_FILE_WRITE	MPI_FILE_READ_ALL MPI_FILE_WRITE_ALL
	nonblocking & split collective	MPI_FILE_IREAD MPI_FILE_IWRITE	MPI_FILE_READ_ALL_BEGIN MPI_FILE_READ_ALL_END MPI_FILE_WRITE_ALL_BEGIN MPI_FILE_WRITE_ALL_END
shared file pointer	blocking	MPI_FILE_READ_SHARED MPI_FILE_WRITE_SHARED	MPI_FILE_READ_ORDERED MPI_FILE_WRITE_ORDERED
	nonblocking & split collective	MPI_FILE_IREAD_SHARED MPI_FILE_IWRITE_SHARED	MPI_FILE_READ_ORDERED_BEGIN MPI_FILE_READ_ORDERED_END MPI_FILE_WRITE_ORDERED_BEGIN MPI_FILE_WRITE_ORDERED_END

Table 2.1.: Overview of MPI-I/O functions [DF12]

2.1.4. ROMIO and ADIO

ROMIO is a widely used implementation of the MPI-I/O. It is used by MPICH, LAM, MPI-HP, MPI-NEC, MPI-SGI [BISC08] and other implementations. It provides many optimizations for parallel I/O such as two-phase I/O, collective I/O and data sieving. As the implementations of many of these optimization algorithms depend on the capabilities of the underlying file system, ROMIO would have to adopt to each of them for supporting the file system. As this issue is not ROMIO specific and existent before ROMIO, Thakur et. al. introduced ADIO in 1996 [TGL96]. ADIO is an Abstract-Device Interface for Implementing Portable Parallel-I/O Interfaces. It separates the machine-dependent aspects from machine-independent once as shown in Figure 2.4. ROMIO is based on ADIO as it allows to use this helpful separation[TGL99]. Therefore, ROMIO can be seen as an implementation of the machine-independent parts for the MPI-I/O interface. Due to this separation the core elements of the optimization often lay in the ADIO layer as they are dependent on the file system. ADIO is developed as a part of ROMIO, so in this thesis referencing on ROMIO always includes ADIO.

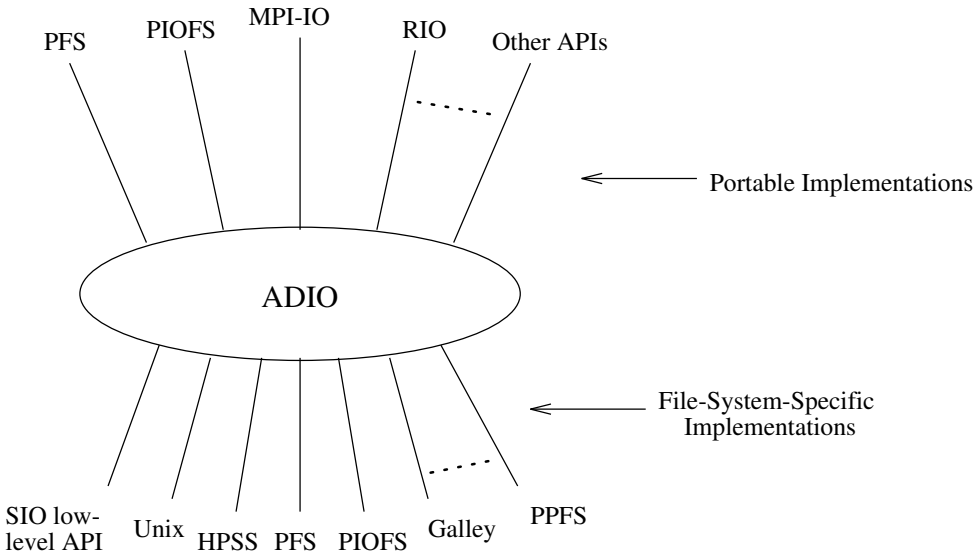

Figure 2.4.: ADIO [TGL96]

2.1.5. Data sieving

Data sieving is a method to increase the performance for non-contiguous data access. As mentioned in Chapter 1 non-contiguous data means data which is scattered over a larger region in a file, interrupted by unwanted data. In this thesis, wanted data will be reference as d_{data} and unwanted as d_{hole} or as data and hole if the difference is clear. For small data sizes and hole sizes the performance is usually bad, as the overhead of a new access for each data block sums up and outweighs the actual time for accessing data. The data sieving method overcomes this problem by accessing large regions of data and holes to a temporary buffer and sieves out the holes. As long as the additional time for accessing the holes is smaller than the overhead time for a new call, this method provides increased performance. In Figure 2.5 the access of two larger regions, first in red and second in blue, is shown. The same data sieving buffer is used for both access. By setting the size of the buffer the maximum overhead of memory use for data sieving can be set. The method of data sieving is initially presented by Thakur et. al. in [TBC+94] and further improved in[TGL99] [TGL02].

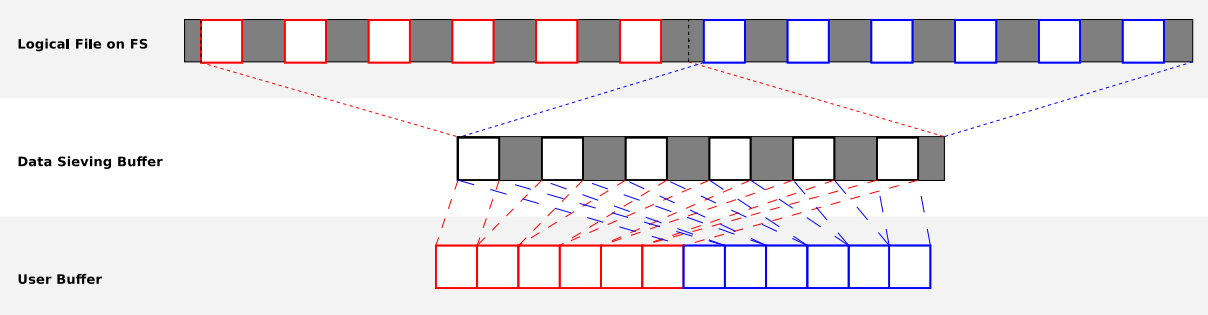

Figure 2.5.: Data Sieving: file to user buffer

2.2. Related Work

As improving I/O performance is vital for nowadays high performance computing there is a lot of research done in this field. Strategies like the two-phase I/O [BdRC93] and collective I/O [TGL99] were developed. Collective I/O on parallel file systems were further enhanced by Zhang et. al. in [ZJD09]. Great collections of the improving techniques were implemented such as the ADIOS library [JKH+08] or the ROMIO [TGL99] implementation of MPI-I/O. But most of the research is targeting the collective and parallel aspect of improving data access. The research on optimization of non-contiguous access with methods like data sieving is quite small. Except for the introduction of data sieving in [TBC+94] and its further improved [TGL99] [TGL02] and integration to ROMIO there were two new ideas of improving data sieving. In [CLA+14] Chen et. al. introduced a performance mode for data sieving. This improves the current state of data sieving by turning it on and off depending on the pattern which is accessed. To decide this the performance of the underlying system is used. The other idea is to improve the performance of non-contiguous I/O by using machine learning tools to determine the best access method based on pattern and system parameters. So done by Schmidtke in [Sch14] and Kunkel et. al. in Predicting Performance of Non-Contiguous I/O with Machine Learning [KMB14].

3. Design

This chapter delivers a solution to the problem of accessing non-contiguous data in an efficient way. The process of finding a solution for this problem is demonstrated in Section 3.1 while explaining the obstacles faced on the way to a detailed implementation. In Section 3.2 these solutions are concluded and goals for a new implementation are proposed. Afterwards, an overview of my NCT library, which implements the solutions, is presented in Section 3.3.

3.1. Discussion of data-sieving algorithms

Improving the current possibilities of accessing non-contiguous data, requires to analyze the current solutions first and gather ideas for advanced solutions. In general, optimizing data access is difficult, because one needs knowledge of future spatial and temporal access pattern. Without this knowledge only prediction is possible, like for example done on hard disk drives for the read-ahead mechanism. Implementing the optimization on a high level, close to the user, can provide this knowledge due to the fact that the user is familiar with the access patterns of his program. For accessing data the user usually knows which data he wants to access, so there should be an easy possibility to share this information with underlying software. For example, this is done by the MPI file view. This great concept of using knowledge of data pattern is essential of optimizing non-contiguous data access with methods like data sieving.

ROMIO is a widely used implementation for MPI-I/O which offers data sieving. The data sieving is parameterized by the buffer size and the option to turn it on or off, both to set by the user. If it is on, all access is done in blocks with a size of the buffer, starting with an offset to the first data block. It does not matter if the additionally accessed part hits more data blocks or not. Hitting more data blocks is the desired case where the performance improvement is gained. So the user has to know, whether data sieving is helpful with this specific data and therefore activate it conditionally. If the variety of the data pattern is so diverse that there are parts which benefit from data sieving and those which do not, the user either has to split up the access or decide to have the inferior way to access some parts of the data. Therefore, the first thing to improve is that the data sieving should automatically turn off in cases where its not beneficial or even harmful. The easiest example for a harmful pattern is the case where one block of data is accessed and all additionally accessed data is hole data, which is not needed. In that case the algorithm should access only the desired data block, without accessing additional hole data.

The first problem to solve in new and more advanced data sieving solutions is to automatically determine if data sieving, over the next hole is beneficial or not. To achieve

this, at least two additional information are needed by the data sieving algorithm: Firstly the pattern of the data and secondly the maximum size of a data hole, which could be accessed additionally and still improve the performance. This maximum size of data holes on the other hand depends on the performance of the file system, which could make determination of this parameters a challenging task on its own.

For a data access on an single drive which is directly attached to the computer, this maximum hole size would be easy to determine by measuring the bandwidth for data transfer and latency for a new data access. But nowadays clusters now store their data in a SAN whose performance varies on much more parameters. The bandwidth depends on the used network, how many nodes are demanding and how many are delivering data content. The latency is also depending on the network and seek times, which could be hard to determine in such complex storage systems. These are just some factors which affect the maximum data hole size which should be accessed through data sieving. In the following, the set of parameters which are relevant for this thesis are listed. This selection seems to have a major contribution on the performance.

- Network Bandwidth
- Network Latency
- Bandwidth I/O Nodes
- Number I/O Nodes
- Stripe Size

Due to the amount of parameters that could be taken into account and the shared nature of storage due to multiple users in these kind of cluster and SANs, the performance is hard to determine. Therefore, a static value for the maximum hole size will not meet the needs. As a consequence there cannot be a solution simply relying on the latency and the bandwidth. So it appears that advanced data sieving algorithm should be highly parameterizable at run time. With this feature it would be possible to set the parameters according to the status of the whole cluster system. These could be predetermined by machine leaning algorithms, which analyze the performance trace of the cluster. It would be a perfect task for the SIOX architecture developed by Kunkel et al. [KZH+14].

3.2. Designgoals

As introduced by the previous section a new data sieving algorithm should be much more flexible to provide good performance on modern systems. This flexibility should allow the data sieving to adopt to the underlying system by accounting as many parameters as it could get. On the other hand these parameters are not primal meant to be set by each user manually. The desired goal is to provide sophisticated algorithms, tuned to the underlying hardware, to automatically choose the right behavior. This provides more portability as the algorithms can be tuned by the administrators and the user just turn on the optimization. Another major goal was the separation of data sieving and MPI-I/O. Due to the fact that using MPI-I/O may not have spread as wide as it could,

and using standard POSIX calls for I/O is common, providing data sieving and file views at the POSIX layer provides a wider use of data sieving. This would extend the current ability of POSIX calls like readv() and writev(). Nevertheless using MPI-I/O provides great potential to improve performance, therefore the new data sieving should be easily portable to MPI-I/O.

1. **Flexibility**
   - Allow algorithms to adopt to the underlying system
   - Easy to modify behavior of data sieving
2. **Automation**
   - Possibility to use sophisticated algorithms which do not need further user adjustments
3. **Universality**
   - MPI Independence for wider use of data sieving
   - POSIX level API

3.3. NCT Library

The non-contiguous (NCT) library is implemented to achieve this goals. It is a library which provides POSIX like read and write calls on a given file view. These views can be set and changed on run time. To provide the postulated flexibility to adopt to different environments, a modular system for different data sieving methods is provided by NCT. Granting a wide usability of this data sieving library is given by its implementation on the low level of POSIX calls. Transferring this to MPI-I/O is much easier than it would be the other way around. One possibility to integrate NCT on the MPI-I/O layer is shown in fig. 3.1. The design of NCT is presented in this section by first explaining the file view followed by the modular data sieving algorithms as these are the core elements of this implementation.

3.3.1. MPI-I/O integration

Making the new data sieving algorithms available in MPI is needed as MPI-I/O provides many confirmations for parallel data I/O. Also it would be even easier to switch to this new data sieving algorithms as there would be no need to modify existing programs. There are two ideas to make NCT available in MPI. The simple one would be to build some wrapper library around NCT. This library would convert MPIs file view to NCTs and could wrap the NCT functions so they provide the non collective MPI-I/O functions. Implementing collective operations would need much more additionally implementation. That is where the second possibility came in; integration in the ROMIO implementation. This would be more complex as it needs to adopt to the ROMIO code. But on the other hand many functionalities to provide collective I/O could be used. Also optimizing the NCT algorithm for special file systems would be possible due to the separation of file

system depending parts and the independent parts via ADIO. This concept seems to fit NCTs concept to separate the decision part from the accessing part.

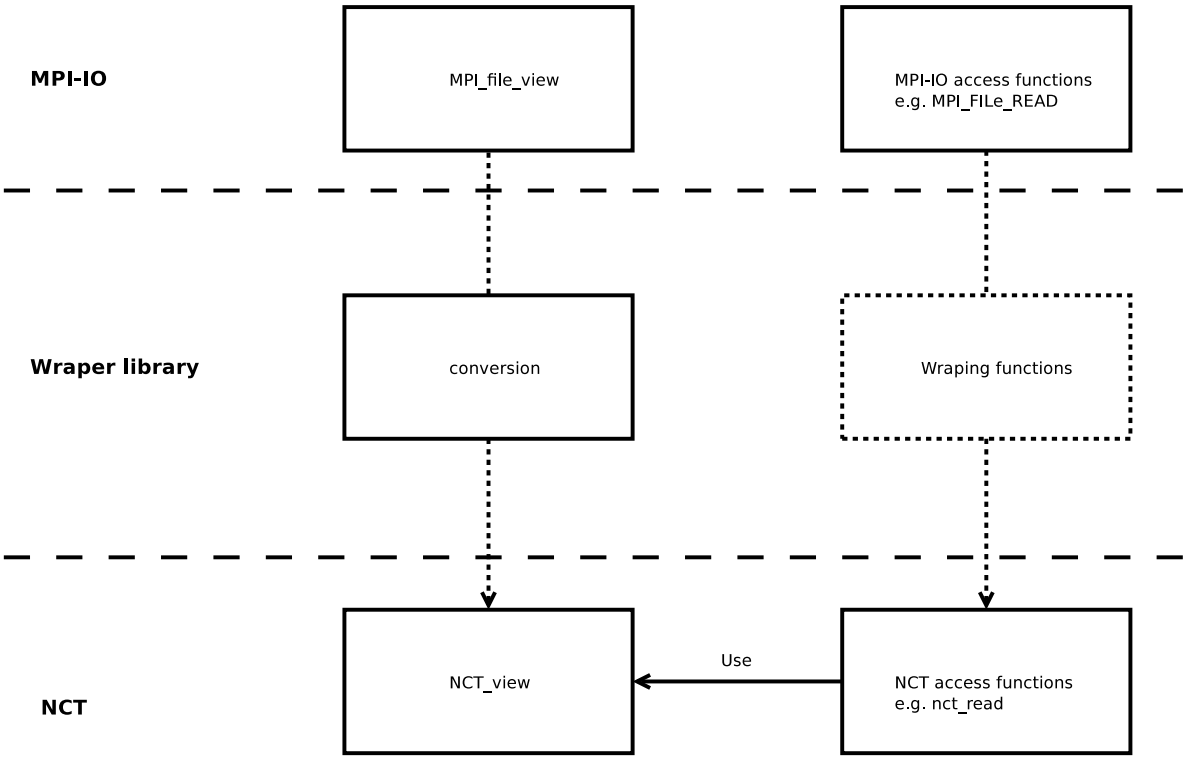

Figure 3.1.: NCT integration in MPI I/O

3.3.2. File View

MPI's file view is a good starting point for developing an own view based access. But MPI's version is based on MPI data types which are not available in that way for standard C programs. So for NCT the view is described by a list of tuples, where each tuple represents a block of data and the delta offset to the next tuple instead of data types. This allows to build up a view, which is as generic as it can be. To skip data in the beginning of a file, e.g. headers, there is also an initial offset to the first tuple which can be set as shown in figure Figure 3.2. Another common case should be that the pattern on the file is repetitive, for example reading every second entry from an array. Creating such patterns is easy because of the feature to start with the first tuple over again if the access goes further than the last tuple. The initial offset is only relevant when determining the offset to the first tuple.

With the description of the data layout by the tuple, the NCT view is defined. There are some more information for the optimization, such as the size of the data sieving buffer. In MPI-I/O the information about the size was sufficient. For NCT, the decision was made that the user should provide and allocate the data sieving buffer. This gives the advantage of letting the user decide how to handle buffer access in multi-threaded environments. Also the data sieving buffer is located in the user's memory and not in

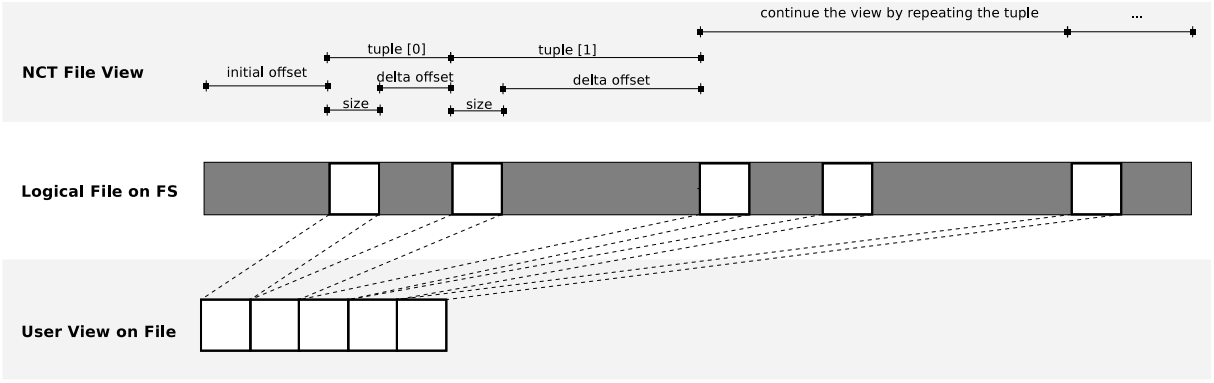

Figure 3.2.: NCT File View

the library's, which should grant better performance for copying data between users and data sieving buffer. Therefore, not only the size but also a pointer to allocated memory must be provided to create a view. After this setup phase it would be possible to perform data sieving based on the view. The goal was to have more parameters to adjust the behavior of the data sieving. For all this parameters a struct, called info, which holds all this additional information, is also given to create the view. Setting up multiple views in the beginning allows an easy access to complex data structures in following code as the setup only needs to be done once.

3.3.3. Modular data sieving algorithms

Making the NCT library flexible to use, seems to be a very desirable goal, due to the fact that data sieving is depending on the system it is used on and so should be adapted to it. To make the NCT library as flexible as needed a modular system was designed, which allows to separate the view creation as well as the data copying between buffers form the actual decision when and how to apply data sieving. To achieve this the decision was broken down to the question whether it is faster to access the next data hole or trigger a new access call. Thus, a data sieving algorithm in NCT is implemented by a simple true or false decision. These decisions are encapsulated as functions called `aggregate functions`. This name was chosen due to the fact that their decision determines if the next tuple will be aggregated to the next access on the file system. Implementing different or new data sieving algorithms with NCT came down to implementing one function, to return true or false. This opens even the possibility to let the user provide an aggregate function to the NCT library on view creation and therefore make it highly customized to the user's and system's needs. It is also possible that the administrator of the system provides an aggregate function to the users so they can profit from a well adopted data sieving by just turning it on.
Evaluating different data sieving algorithms to improve them, like done for this thesis, was much easier with this flexibility of NCT. The same benchmark code could be used for different data sieving algorithm by just using one different aggregate function. The different algorithms implemented for this thesis are presented in the following.
Above the decision whether to aggregate the next tuple the aggregate function can also set how many bytes of data should be accessed on the current tuple or how many additional

hole data should be accessed. More details on this is provided in Section 4.4.

Naive Data Accesses As the most basic capability of the NCT library the naive data access, which is to access each data block on its own, bases on the given file view. It gains no improvements on the performance but only provides the use of a data view to hide the holes in a data layout to the user. Evaluating its performance leads to a valuable reference for the other benchmarks, which was the main reason to implement it. Furthermore, this functionality is needed anyway and provided a good base to start at.

ROMIO As a second algorithm the ROMIO implementation, based on the initial MPI-I/O data sieving [TGL99], was chosen. This was done to find and verify the case where a new data sieving algorithm should improve.

Simple Performance Model This algorithm is able to decide whether to use data sieving or not, based on two parameters. First, the bandwidth of the whole system for accessing data and second the network latency which is considered as the main factor for the overall latency on new access calls. These two parameters are chosen as they have a major impact on the performance as shown by Kunkel in [Kun13] [Kun06]. This is also verified by a model of the expected performance, which is further introduced in 5. In cases where data sieving is turned on, the ROMIO like version is used. In a similar way Yong Chen et al. presented such an approach in Performance model-directed data sieving for high-performance I/O [CLA+14] as mentioned in section 2.2.

Extent-based Data Sieving This algorithm decides the same way as the Simple Performance Model if data sieving is beneficial. It is slightly improved by not always filling up the buffer with hole data but only accesses the extent from the first to the last tuple.

Scanfile The scanfile algorithm accesses the entire file starting at the first data block. Additionally, it aligns the data accesses to the given stripe size. This was implemented for a wider base of performance values to compare to.

Advanced Data Sieving Model The advanced data sieving algorithm combines the result of the evaluated data sieving algorithms, to make more sophisticated decisions. To do so the following parameters are considered.

- Bandwidth
- Latency of the network
- Latency of the system calls
- Latency of file system
- Number of I/O nodes
- Stripe size of the I/O nodes

- Block alignment

The bandwidth could be split up in file system bandwidth and network bandwidth, but due to the fact that in the test environment used for this thesis the maximum bandwidth was always set by the limits of the network just one parameter for the maximum bandwidth is used. The latency is given by three main factors, the network latency, the file system latency, which could be also depending on the hole size as seek times on hard disk drives depend on that, and as a minor factor the system call latency. An new factor, not considered by any available data sieving implementation, is block and stripe alignment. This improves performance for multi node storage systems.

4. Implementation

The core topic of this chapter is how NCT is implemented. Presenting the API and structure of the code is followed by some problems and limitations as well as some possible improvements. All along the relating parts form the code.

4.1. API

The API was a good point to start with the implementation as it provides a good overview of what variables are required for the code. Having all the variables from the function signatures, gives some impressions of the code structure. As this impression helped on developing, it will do so for introducing the code in this chapter. Two parts of the API, the file view and the info struct, are previously mentioned in Section 3.3.2. Their signature can be seen in Listing 4.1. The struct and typedef for the tuple are shown in line 1 to 5 and 21 of Listing 4.1. As the `nct_creat_view()` returns a void pointer, typdefed to `nct_view` for hiding the internal structure from the user, a function to properly free this memory is needed. In line 28 `nct_destroy_view` is shown, which provides this functionality. The functions to actually access data are called `nct_read()` and `nct_write` and can be seen in line 30 and 32. Their signature is the same as the `pread()` and `pwrite()` provided by POSIX, except from a additional parameter to hand over the view. This design allows to create multiple views and used them depending on the situation.

```
1  struct nct_tuple_t
2  {
3      uint32_t size;
4      uint32_t deltaOffset;
5  };
6
7
8  struct nct_info_t
9  {
10     double latencyFS;
11     int bandwidth;
12     int numNodes;
13     double latencyNetwork;
14     uint64_t stripeSize;
15     uint64_t blocksize;
16     int dsMethod;
17 };
18
19 typedef struct nct_tuple_t nct_tuple;
20 typedef struct nct_info_t nct_info;
21
22 #ifndef NCT_INTERNAL_HPP
23 typedef void * nct_view;
24 #endif
25
26 nct_view nct_create_view( uint32_t initalOffset, size_t ds_bufsize, void * ds_buf, int
       tupleCount, nct_tuple* list, nct_info* info);
```

```
27
28 void nct_destroy_view(nct_view view);
29
30 size_t nct_read(int fd, void* buff, uint64_t offset, size_t byte_count, nct_view view);
31
32 size_t nct_write(int fd, void* buff, uint64_t offset, size_t byte_count, nct_view view);
```

Listing 4.1: nct.h

4.2. Structure of the Code

The code is split into the following files.

- **nct.c**
  - This file contains the different aggregate functions, the view creation and destruction as well as wrapping for the internal functions to provide a straight interface to the user.
  - New aggregate function should be added in this file as it provides a good overview of the different capabilities.
- **nct.h**
  - This header file defines the actual API and some needed data types, like the info struct and the nct_tuple struct.
- **nct_internal.c**
  - This file contains the function which actually aggregates the tuples and accesses them trough the data sieving buffer.
- **nct_internal.h**
  - This header provides all the internal data types and structs. It also declares the functions for nct_interal.c of the nct.c file.

The distribution of the functions in nct.c and nct_internal.c represents the separation of the data accessing part of data sieving and the decision part whether further action is useful.

4.3. File View

The file view is a core element of NCT as it holds all the information needed to perform data sieving. As described in Section 3.3.2 the data sieving buffer and the info struct is handed over as well as the list of tuple. The list of tuple is internally copied to a new one with some additional information. For iteration and search on the tuple list it is handy to know the total extent of the view and the offset of each tuple to the begin of the view. These information are added to the internal tuple list while copying the user provided list. In Listing 4.2 the internal tuple and view struct is shown. For each tuple the accumulated

size and offset are stored. They represent the view and the logical offset from starting at the first tuple. In the struct `view` the total extend of all tuple and the data it holds is stored on typeSize and tupeExtend. The other fields of the view struct are known from the view's creation, introduced in Section 3.3.2.

```
1  struct nctTupleInternal
2  {
3      uint64_t size;
4      uint64_t deltaOffset;
5      uint64_t cummulativeOffset; // total offset form view beinning, without inital offset
6      uint64_t cummulativeSize;   // total size of data in till her, includeingf this
           element
7  };
8
9  struct nct_view_t
10 {
11         void *      ds_buf;
12         size_t      buffsize;
13
14         uint64_t    initalOffset;
15         int         tupleCount;
16         struct      nctTupleInternal * tuples;
17         uint64_t    typeSize; // bytes of data the type holdes
18         uint64_t    typeExtend; // bytes type ocupies on file (holes + type size)
19
20         struct      nct_info_t * info;
21 };
```

Listing 4.2: Internal tuple an view struct

4.4. Access Functions

The access functions are implementing the actual data access on the file. They have to access the data considering the given view. For NCT there are two main access functions. Both are capable to read and write data. Therefore, internally `nct_read()` and `nct_write()` both use this function in the respective mode.

The first function provides access to the data on the given view by skipping the holes and makes individual access to the data blocks. The naive algorithm is implemented by this if its used directly at the file. As this algorithm is also needed for data sieving to access the data in the data sieving buffer, this function can do both. Accessing file pointers as well as the data sieving buffer. In case of reading, from file or data sieving buffer to user buffer and for writing from user buffer to file or data sieving buffer. As this function access each tuple individually it is named `_nct_access_tuple()`.

The second function is needed to perform data sieving. For NCT data sieving is modeled as an aggregation of tuple. As mentioned in Section 3.3.3 this aggregation is controlled by the aggregation functions. These determine if its beneficial to aggregate the next tuple and therefore the delta offset of the current tuple. The major task of the second function is to iterate over the tuple and ask for every one if the next one should be aggregated, too. If the next tuple should be aggregated, all aggregated tuple including the holes are accessed through the data sieving buffer. In case of reading this means to read them via `pread()` to the data sieving buffer and then call the `_nct_access_tuple()` function to copy the data from the ds buffer to users buffer. In case of write the data is also read to the data sieving buffer then the data is modified by calling the `_nct_access_tuple()` to

write the users data to the data sieving buffer, which overwrites only the tuple and not the holes. Afterwards the data is writen back to the file system by calling `pwrite()`. As this function access aggregated tuple it is named `_nct_access_aggregated_tuple()`.

The prior introduction should provide an impression on the mechanics of accessing non-contiguous data with NCT. As there are many problems due boundary conditions and requirements to provide the desired flexibility of NCT both algorithms get much more complex, especially `_nct_access_aggregated_tuple()`. A closer view is now given by addressing the issues on by one.

First Tuple Getting the tuple to start is a problem both algorithms are facing. As the user could access data at any view offset the associated tuple has to be found. This is done by a binary search on the tuple list. The additional information on the tuple generated on view creation make this an easy task. As the view could be repetitive the number of full views must be subtracted from the view of set. The typeSize and typeExtent came in here. All this is provided by the viewAdrToLogicAdr() function which signature is shown in Listing 4.3. The function determines the logical Offset, the current tuple number and the bytes to access on the current tuple for a view offset on a given view. The amount of bytes to access on a tuple is important as the user could start accessing just in the middle of a tuple.

```
1 static void viewAdrToLogicAdr(uint64_t * logicalOffset, int * curTupleNum, uint64_t *
      bytesToAccOnTuple, uint64_t viewOffset, nct_view view);
```

Listing 4.3: find

Repetitive views As repetitive views are possible and likely to be the common case, starting over with the the first tuple is an essential feature. To provide a wrap around to the first tuple a linked list could have been used. But this would cause some unnecessary overhead because C does not support such a structure by default. Storing the tuple in a simple list implemented by an array and set the tuple number to the first, if proceeding from the last to the next is easy and fast and therefore implemented on both access functions in this way.

End of access The end of accessing is usually reached when all demanded data is accessed. But there are some exceptions and some special cases for this, too. Like for accessing the first tuple, it could happen that the user wants to access just till the middle of a tuple. Therefore, on each tuple a check is needed to verify it has to be accessed completely. Another exception is that the file could be smaller the the users request. On this case the `nct_read()` and `nct_write()` should return how many bytes are actually access, like the POSIX calls does. This also concerns both access functions.

These boundary conditions need to be concerned to access data through a view. The following conditions needs to be observed to perform data sieving in general or due to the way its implemented on NCT.

Data sieving buffer fill level One of the general conditions for data sieving is that aggregating more tuple for the next access needs to stop before it is too large for the data sieving buffer. This makes it one of two condition in `_nct_access_aggregated_tuple` which trigger an access regardless of the aggregate functions decision. The other one is the end of access, due to the amount of predigested by the user.

File locking on writes For writing data through a data sieving buffer, a read-modify-write access is needed. During the hole time the region in the buffer need to be locked on the file system. This is implemented by calling `flock()`. As `flock()` is an advisory lock parallel file systems may handle the locks different.

Additional options for aggregate functions A major goal was to split the accessing part of data sieving from the decision part. With the previous condition implemented in `_nct_access_aggregated_tuple()` the accessing part is done. To enable the aggregate functions to control the behavior appropriate only deciding weather or not to aggregate the next tuple is not enough. Sophisticated data sieving needs more parameters to control the behavior. Three more optional parameters were implemented to provide this.

> **Bytes to Access on current tuple** As it could be beneficial to stop accessing in the middle of a tuple the aggregate function should have the possibility to decide how much of the current tuple should be accessed.
>
> **Hole bytes to aggregate after current tuple** Also the opposite of the previous case could be needed. So aggregating some hole bytes additionally after the current tuple should be also possible.
>
> **Hole bytes to aggregate before next tuple** Third there is the possibility to add hole bytes before the first tuple of the next access through the ds buffer.

Implementing these control parameters, also needs to consider all the other boundary conditions explained earlier. This results in even more complex offset handling and made some limitations to the control parameters. Especially the parameters to add hole data. They will only be minded if the data on the current tuple is accessed completely, as it does not make sense to access any hole data if there is data left on the current tuple. The amount of hole data which could be accessed additionally, no matter if behind the current or before the next tuple, is limited to the hole size of the current tuple. A similar limitation is given to the amount of data which should be accessed on the current tuple. This can not be more than the tuple size. As it could be the case that the current tuple is only be accessed partial, e.g. to end of file or a filled data sieving buffer, the user gets the maximum amount which is accessible. Also non of the parameters have any effect if the next tuple will be aggregated as this means that the whole tuple, data and hole, will be aggregated.

The code can be found in the appendix. It is not that well structured because of having all these condition in `_nct_access_aggregated_tuple`, made keeping track of the offsets a challenging task. Keeping the implementation clean and readable were always minded but due to time limitations not the major goal. Refactoring is planed for future work on NCT.

4.5. Aggregate functions

The aggregate functions define the behavior of the data sieving as introduced in Sections 3.3.3 and 4.4. In this section, they are explained on detail by showing the ROMIO and a simple performance model implemented as aggregate function. In Listing 4.4 the signature of a aggregate functions can bee seen. The return value which determines if the next tuple should be aggregated and the three option parameters introduced in Section 4.4 are the 4 possibilities to control the data sieving behavior. All the other arguments of the function are information about the current state of the aggregation which could be use for decision. At the current state the information if it is a read or write access is missing. The importance of this, shown up on the benchmark which will be discussed in Chapter 5. Adding this is planned for future work.

```
1  static int aggregate_romio(
2      uint64_t * curLogicalOffset,
3      uint64_t curViewOffset,
4      uint64_t viewBytesToAccNext,
5      uint64_t logicalBytesToAccNext,
6      int curTupleNum,
7      uint64_t * bytesToAccOnTuple,
8      uint64_t * holeBytesToAggrOnTuple,
9      uint64_t * holeBytesToAggrOnNextAcc,
10     nct_view view
11     )
12 {
13     size_t dsBufSize = view->buffsize;
14     struct nctTupleInternal * tuple = view->tuples;
15
16     if(tuple[curTupleNum].deltaOffset  >= dsBufSize - logicalBytesToAccNext)
17     {
18         *holeBytesToAggrOnTuple = dsBufSize - logicalBytesToAccNext;
19     }
20
21     return 1;
22 }
```

Listing 4.4: how to access tuple from memory or file

ROMIO The ROMIO algorithm always fills up the data sieving buffer no mater if it hits another tuple or not. Implementing this behavior as an aggregate function is simple. As a first point the function should always return 1 to aggregate as many tuple as it can get. As the `_nct_access_aggregated_tuple` function access automatically if the data sieving buffer is full there is no need to ever return 0. But to fill hole bytes after the last tuple which fits in the data sieving buffer, the `holeBytesToAggOnTuple` variable needs to be set to the amount of the remaining space in the data sieving buffer. This can be seen in Listing 4.4 line 16 to 19.

Simple performance model The simple performance model decides the aggregation depending on the time needed to access the hole compared to the time over head for a new access. To calculate the times the overall bandwidth, the latency of the network and the file system is used. Like the ROMIO aggregate function this is also easy to implement. The information about the system are stored in the info struct, provided by the users application. As shown in Listing 4.5 line 24 and 25 it is only needed to calculate and

compare the two times to decide whether or not to aggregate the next tuple.

```
static int aggregate_simpel_pm(
    uint64_t * curLogicalOffset,
    uint64_t curViewOffset,
    uint64_t viewBytesToAccNext,
    uint64_t logicalBytesToAccNext,
    int curTupleNum,
    uint64_t * bytesToAccOnTuple,
    uint64_t * holeBytesToAggrOnTuple,
    uint64_t * holeBytesToAggrOnNextAcc,
    nct_view view
    )
{
    double timeAccHole;
    double timeNewAcc;
    nct_info * info = view->info;
    struct nctTupleInternal * tuple = view->tuples;
    int bandwidth = info->bandwidth;
    double netLatency = info->latencyNetwork;
    double fsLatency = info->latencyFS;
    struct nctTupleInternal curTuple;

    curTuple = tuple[curTupleNum];

    timeAccHole = (curTuple.deltaOffset / (double) bandwidth) ;
    timeNewAcc = (netLatency + fsLatency);

    if(timeAccHole < timeNewAcc)
    {
        *holeBytesToAggrOnTuple = 0;
        return 1;
    }
    else
    {
        *holeBytesToAggrOnTuple = 0;
        return 0;
    }
}
```

Listing 4.5: Simple performance model

Advance data sieving Due to time limitations implementing a new data sieving aggregate function which address the flaws of the other algorithms, was not possible. In Chapter 5 the observed characteristics were evaluated and applied to the design ideas for the advance data sieving from Section 3.3.3.

5. Evaluation

This chapter is about the different data sieving algorithms' performance and why there are certain characteristics. First, the benchmark program as well as the two cluster systems used for benchmarking are introduced.Afterwards, the results of the improved algorithms are discussed, from no data sieving over two simpler algorithms to the solution of this thesis.So the improvements in contrast to the previous algorithms can be explained. For conclusion all benchmarks are quickly summarized.

Proper benchmarking is a hard task to accomplish and for wider use of the results, it would be great to use standardized benchmarks. This goal is difficult to combine with the special needs for the program which is to benchmark and often a smaller amount of time is necessary to build an own benchmark program. So done in this thesis.

5.1. Benchmark Tool

The implemented benchmark, called NCT-bench, should access a file by using the NCT library and measuring the data throughput. To avoid writing two programs it is used as well as a test program to debug the library. To be flexible in the way of using this tool, it should provide as many parameters to tune as possible. As noticed later in progress there should have been used more of them. That had made write the jobs scripts and plotting the data much easier.

NCT-bench starts checking, if there is a file of the correct size to run the benchmark and precreates an empty file, with the option to disable the precreation and benchmark the performance on writing a new file. For running the actual data access, the NCT library needs a view on the file. NCT-bench creates the view based on following parameters:

- Size of the data sieving buffer
- Initial offset
- Data pattern as tuple
- Data sieving mode
- Bandwidth (optional. only used by some methods)
- Network latency (optional. only used by some methods)
- File system latency (optional. only used by some methods)
- Stripe size of I/O server (optional. only used by some methods)

- Number of I/O server (optional. only used by some methods)

The optional parameters, like bandwidth, network latency, and file system latency, given to the info struct are currently hard coded to the benchmark.
After setting up the view, the file will be accessed by the benchmark tool in blocks of 30 MBytes of data until either the complete file is accessed or more then 100 seconds passed. This should be enough to benchmark the data throughput properly. At this point should be mentioned, that the 30 MBytes chunks of data are the data without the holes from the file. With small data block and big holes in a pattern, accessing large files completely could be accomplished really fast. The elapsed time is measured with clock_gettime().

5.2. Test system

There are benchmark results from two system used and discussed in this thesis. For one the cluster from the working group of scientific computing (WR-Cluster) were used. Which was been used also by Daniel Schmidtke in his work of analyzing data sieving [Sch14] and helped for further understanding on data sieving in ROMIO. Dr. Kunkel executed the benchmarks also on DKRZ porting system for their new supercomputer HLRE3, so the specifications of both systems are introduced. As some effects can be seen better on the results form the DKRZ cluster, there are some graphics shown from these benchmarks.

5.2.1. WR Cluster

The WR-Cluster is a small system used for research and teaching. There are different kinds of nodes. The ones used for this thesis are the Westmere and Sandy Bridge based ones. Ten from both of them are available in the system. The Westmere are the main compute nodes in the cluster and used by a larger group of people. To run the benchmark program one of these nodes was used. The Sandy Bridge nodes are used less for computing because they are configured to act as I/O nodes and therefore are equipped with Lustre. As far as possible it is ensured that there was no other usage of the Lustre storage and Sandy Bridge node while benchmarking. Due to the observation made,there was none, but it should be mentioned as a possible source of measurement errors. All nodes are connected via Gigabit-Ethernet (1GbE) as visualized in 5.1. The cluster uses SLURM for batchqueueing. In the following, there are also some more details about the used nodes.

Network

- Standard: Gigabit-Ethernet (1GbE)
- Bandwidth: 118 MiB/s
- Latency: 0.08 ms

Westmere Nodes

- Processor: Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz
- Memory: 12 GiB RAM
- Operating System: Ubuntu 12.04
- Kernel: Linux 2.6.32

Sandy Bridge Nodes

- Processor: Intel(R) Xeon(R) CPU E31275 @ 3.40 GHz
- Memory: 16 GiB RAM
- Harddisk: WDC WD20EARS-07MVWB0
- Operating System: Centos 6.5
- Kernel: Linux 2.6.32

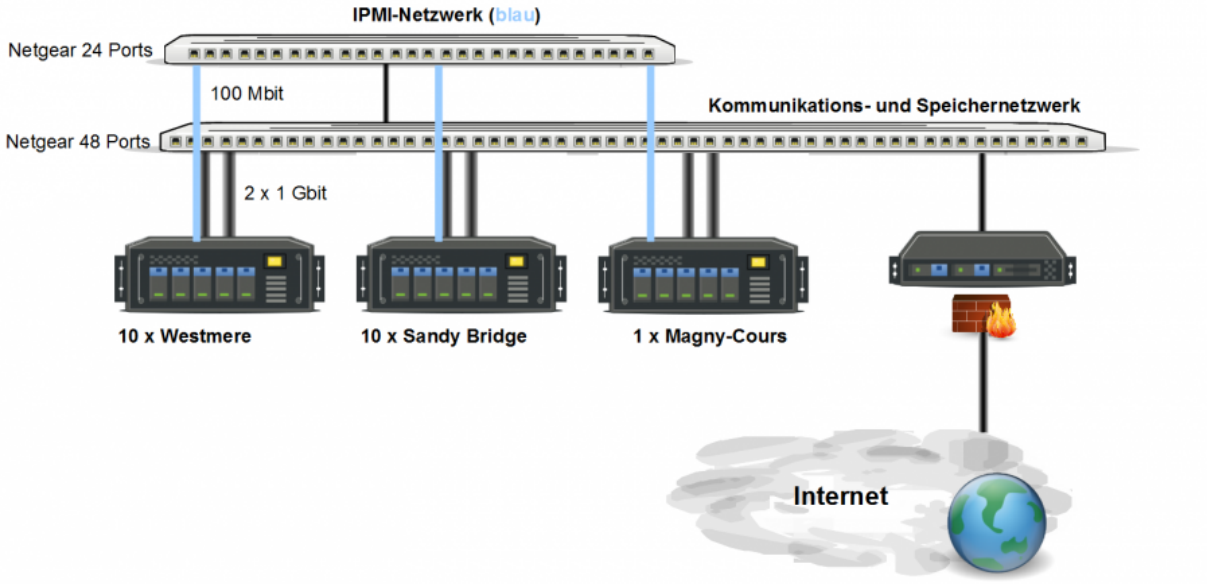

Figure 5.1.: WR Cluster[WR13]

5.2.2. DKRZ Cluster

The DKRZ runs a test system to prepare for their next supercomputer that will be installed early in 2015. The conducted measurements on this porting system were made to study whether the methodology can be applied to learn appropriate Lustre settings and double check if Lustre specific characteristics can be found on both systems. The test system consists of 20 compute nodes and a Lustre 2.5 file system hosted by one ClusterStor 6000 enclosure (SSU) from Seagate with two OSS servers and 84 HDDs. All nodes are interconnected with FDR-Infiniband.

Network

- Standard: FDR-Infiniband
- Latency: 0,001 ms

Compute Nodes

- Processor: Intel E5-2680 v3 @ 2.50GHz
- Memory: 128 GiB RAM

I/O Nodes

- Nodes: 1 x ClusterStor 6000
- File system: Lustre 2.5
- Number of HDDs per Node: 84

5.3. Conducted Experiments

With the benchmark tool several data sieving algorithms, implemented as aggregate functions, were tested to find or confirm weaknesses on them. These benchmarks are done on a 10 GByte file on the Lustre server with one, two, five and ten Lustre nodes once with each 128 KiByte and 2 MiByte stripes. The access patterns are a combination of a data block followed by a hole of not accessed data. This is a pretty simple pattern, which can be benchmarked with a wide variety of hole and data sizes, so it results in an overview of the performance. For this benchmarks the following data and hole size are used in all combinations and all Lustre settings.

Data Sizes:
8, 64, 100, 1000, 4096, 32768, 100000, 1000000, 2097152

Hole Sizes:
0, 8, 64, 100, 1000, 4096, 32768, 100000, 1000000, 2097152, 10000000

The initial offset is set to the hole size used in the specific turn. So pattern always start with a hole to access the file.
Some benchmarks are computed also with an other sets of data and hole sizes, which is of the kind that it is always aligned to the Lustre stripes and by using half a strip size as initial offset always unaligned. The data and hole sizes are just multiples of the stripe size and for the unaligned case are initially offset in the beginning of the file by half of the stripe size. The multipliers are the following.

Data Sizes:
1, 2, 5, 10

Hole Sizes:
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

The benchmarks are done to determine the effect of aligning access to the stripe size of the file system and so if it should be relevant to an advanced data sieving algorithm.

Parameter	Values
Number of Lustre server	1, 2, 5, 10
Stripe size on Lustre server	128 KiB, 2MiB
Access	Read, Write
Initial Offset	Hole size
Data size in Byte	8, 64, 100, 1000, 4096, 32768, 100000, 1000000, 2097152
Hole size in Byte	0, 8, 64, 100, 1000, 4096, 32768, 100000, 1000000, 2097152, 10000000

Table 5.1.: Varied Parameter for Benchmark

Parameter	Values
Number of Lustre server	1, 2, 5, 10
Stripe size on Lustre server	128 KiB, 2MiB
Access	Read, Write
Initial Offset	0, $stripesize/2$
Multiplier for data size	1, 2, 5, 10
Multiplier for hole size	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Table 5.2.: Varied Parameter for aligned and unaligned Benchmark

Having this amount of data is great but also can be a difficult to handle, especially if done for the first time. Proper naming of files for example can make things a lot easier. For getting through all the data, some handy tools from Dr. Kunkel were used. Mostly, they just needed small modifications to fit the new data format. With those tools the data could be parsed from the text file output from each run to an SQLite data base. The data base entries for each Lustre setting were plotted as overview 3D graphs. They show the performance in dependence of the pattern described by data and hole size. Hereby the overall characteristics of the I/O can be seen. The database provides also the possibility to quickly plot detailed views on the benchmarks so interesting points can be shown in more precise ways, like 2D graphs.
The 3D graphs are plotted from discrete point marked on the axis. Points in between are interpolated to create as smooth surface. The Data points are the average of 3 runs, if not explicitly said otherwise. The 2D diagrams often also provide minimum and maximum values and includes the expected performance.

5.3.1. Methodology

The benchmarks were done by using a job scripts, which iterates over all the parameters. On loop for each set of parameter. To have reliable data an outer loop were added to do each benchmark three times. In the inner loop, before starting the NCT-bench, all caches on the node were dropped to ensure that no data form the last benchmark is left in there. After having all benchmark results, they were checked for outliers. The results have to be in a range of 20% from the average. The outliers were benchmark for further validation.

5.3.2. Expected Performance

Before getting actual results, assumptions were made for some characteristics of the results. To validate these assumptions a model of the expected behavior was implemented and plotted. The model is based on the transfer time and the network latency. It provides the a basic characteristics as seen in figure 5.2. For reading access the assumption was made that Lustre reads ahead for the size of on stripe. Also the seek time of hard drives depending on the hole size are included in the reading model. Figure 5.3 shows reading for 128 KiByte stripes and shows slightly less performance than for 2 MiByte stripes in figure 5.4, due to the smaller read ahead size. The model shows that the network latency dominates for accesses which are smaller then 4KiByte. Increasing seek time can be seen in the reading figure 5.3 in the upper right, as the performance goes down with increasing hole sizes.

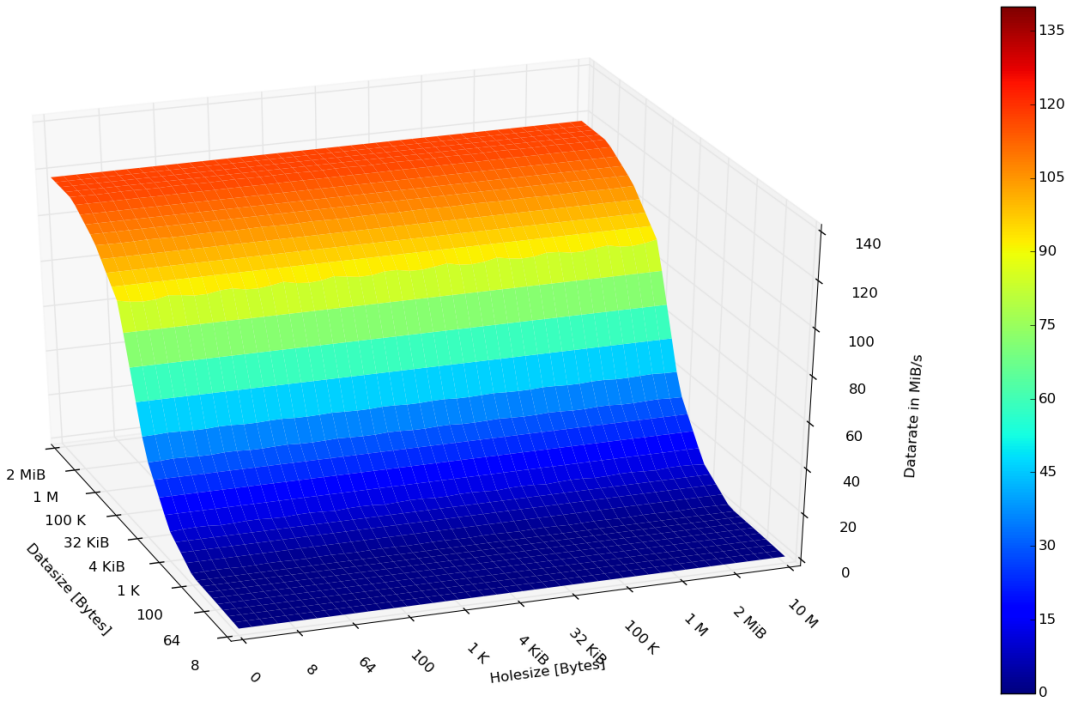

Figure 5.2.: Model WR Cluster for writing

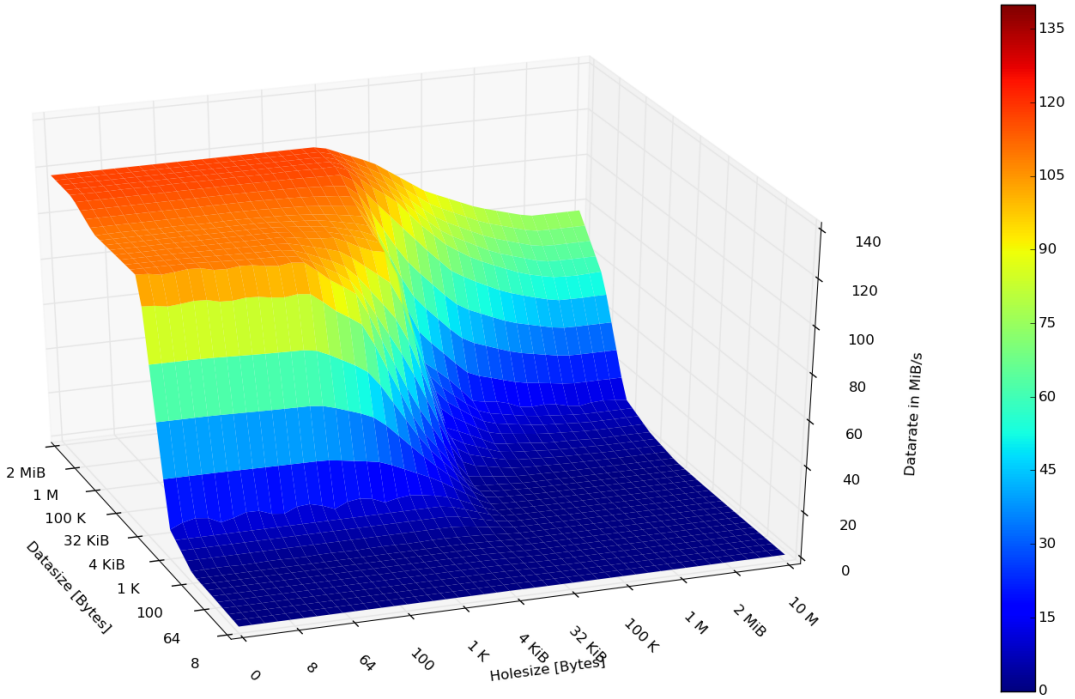

Figure 5.3.: Model WR Cluster for reading with 128 KiB stripes

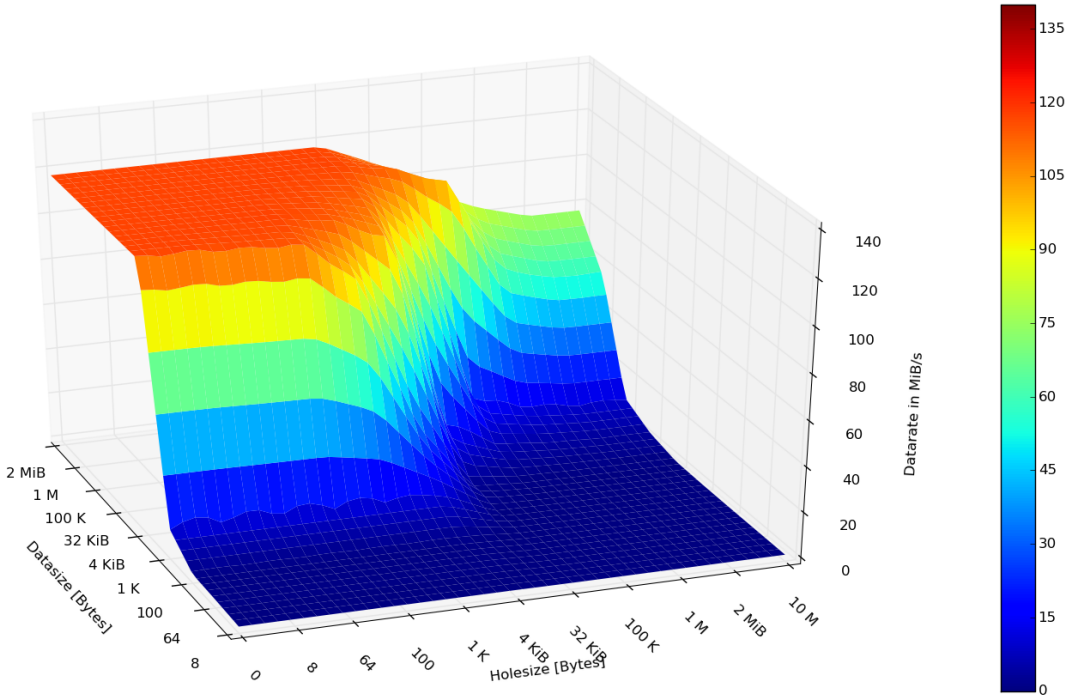

Figure 5.4.: Model WR Cluster for reading with 2 MiB stripes

5.3.3. Naive Data Accesses

The first algorithm is the naive approach to access each data block individually. It is important to understand on which cases a data sieving algorithm should aim to improve the performance. Also it is crucial to know the characteristics of the benchmarked system to be able to interpret the results of the following algorithms.

The first diagram in Figure 5.5 shows the performance for reading from 1 node with a stripe size of 128 KiByte. As expected the performance is pretty good for larger data sizes, more than 4 KiByte, and small hole sizes, less than 4 KiByte. For large holes or small data sizes a bad performance can be observed. This is expected as the disk latency increases and dominates the performance. It is observable for all configurations of Lustre and also on the benchmarks from DKRZ cluster.
Responsible for the good performance are read-ahead mechanisms from Lustre and the hard disks. In Figure 5.6 the effect is more obvious due to bigger steps for hole sizes in the alignment benchmarks. The read-ahead benefits are just given for the 0 byte hole size because 128 KiByte is to much to still benefit or trigger the algorithm. So the big plateau in Figure 5.5 for all the small hole sizes indicates that read-ahead is used and provides good performance for hole sizes up to 4 - 32 KiByte .
Like read-ahead the write-behind effect is also visible in Figure 5.14. The write-behind can not handle small holes like the read-ahead algorithms can. Figure 5.12 leaks therefore the plateau seen for the reading access. The remaining diagram is as expected with a better performance for larger data sets and worse for larger holes sizes. The waves in the front for Figure 5.14 128 KiByte data size may seam unusual, but are easily explained by the fact that for the good performance both server were hit and for the bad only one. This could be better seen in Figure 5.15. For 256 KiByte data and above both server were used all the time. The waves could not be seen in Figure 5.13 as the unaligned cases always hit both server in this case. On the DKRZ Cluster the same effect is visible in Figure 5.16.
Based on this, access in a kind that allows read-ahead or write-behind should be one goal for improved data sieving algorithms.
Another effect which could be seen in Figure 5.6 with access aligned to the RCP size of Lustre; the three points of increased performance in the middle of the diagram. Those are the one where the hole size plus the data size is 1 MiByte and therefore aligned to the size of the lustre RCP. In Figures 5.8 to 5.11 the individual sizes are plotted as 2D diagrams for a more accurate view on the data. Especially if the desired data is always in the beginning of the next RCP. Like for 128 KiByte data and 896 KiByte hole size. In that case for each of the 5 I/O server the first 128 KiByte of every block of 1 MiByte is accessed. This could also be seen as well for the DKRZ cluster in Figure 5.7. As Lustre accesses in RCP sizes this leads to an contiguous read on each server.

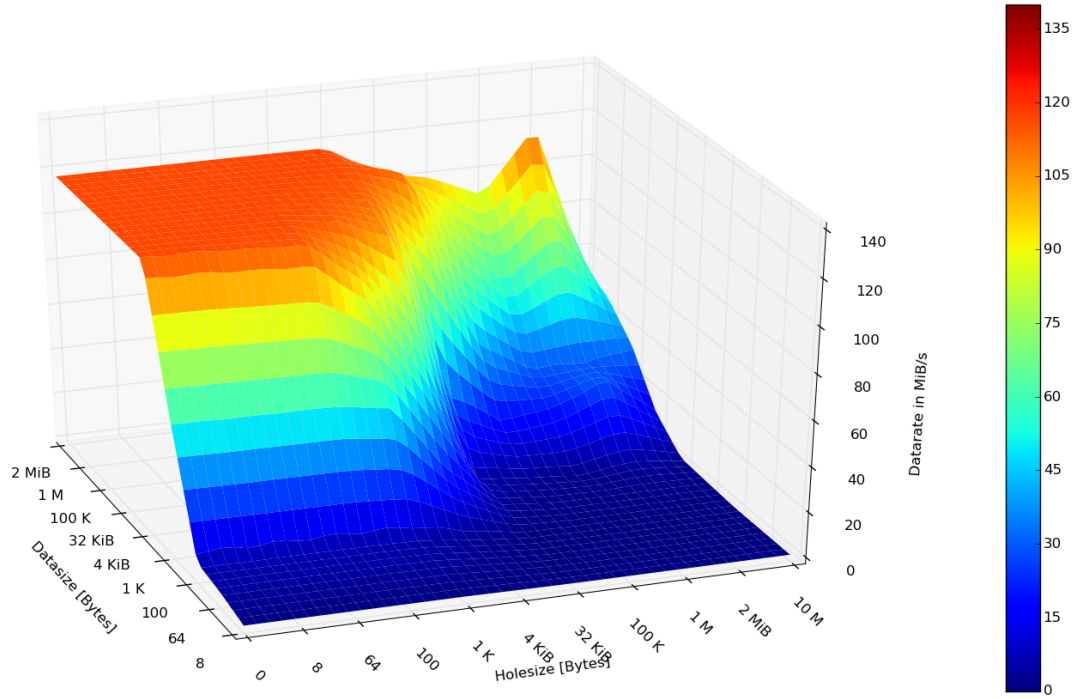

Figure 5.5.: Naive 1 node WR 128 KiByte stripe reading default pattern

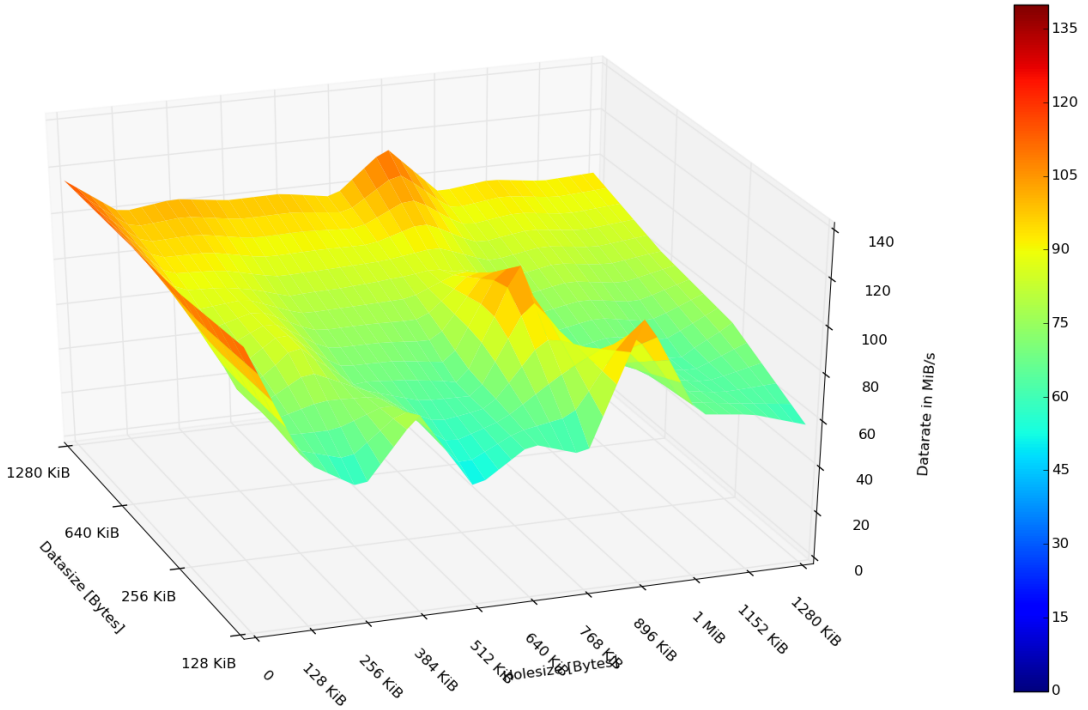

Figure 5.6.: Naive 2 node WR 128 KiByte stripe reading aligned

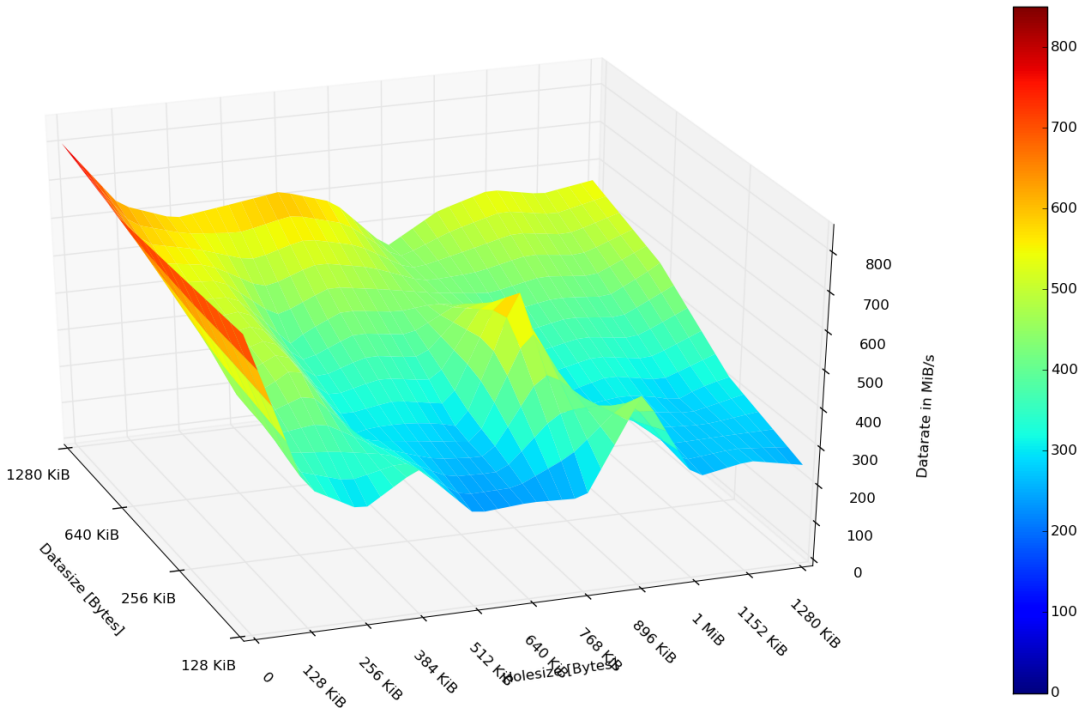

Figure 5.7.: Naive 1 node DKRZ 128 KiByte stripe reading aligned

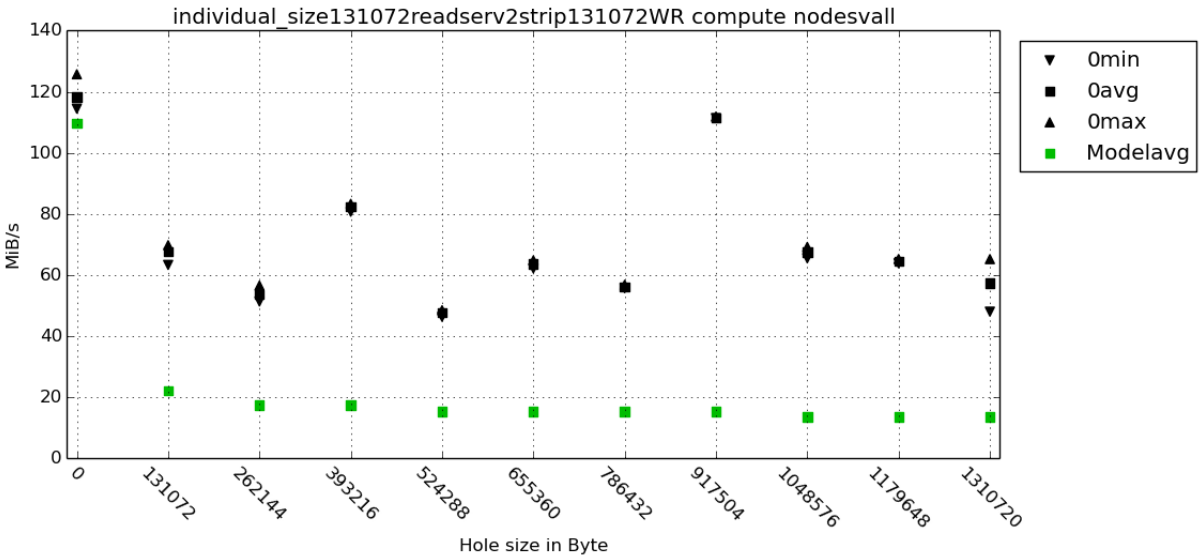

Figure 5.8.: Naive 2 node WR 128 KiByte stripe reading aligned for 124 KiByte data only

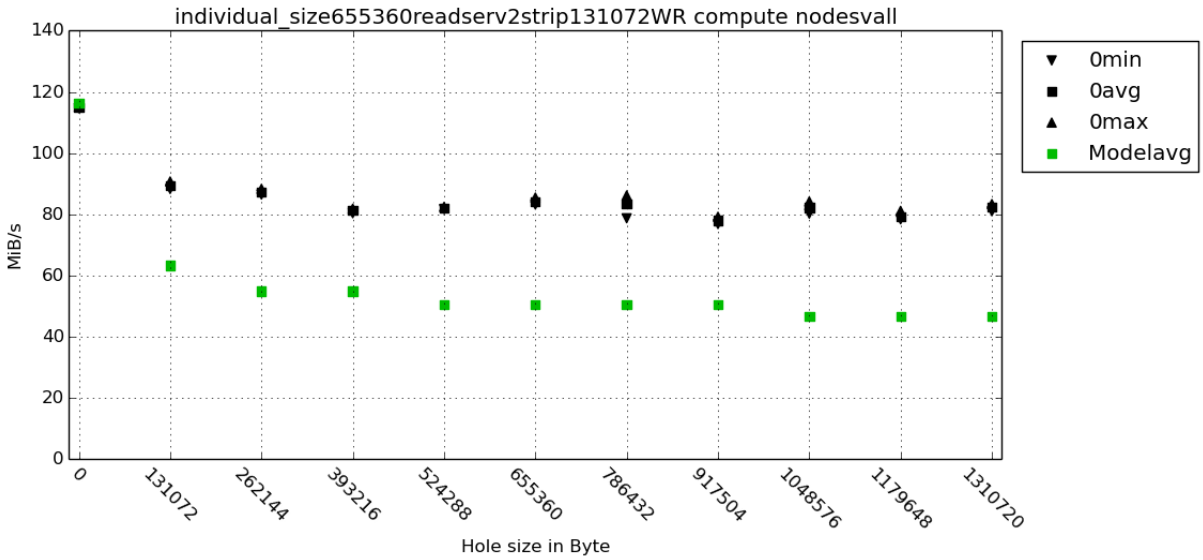

Figure 5.9.: Naive 2 node WR 256 KiByte stripe reading aligned for 124 KiByte data only

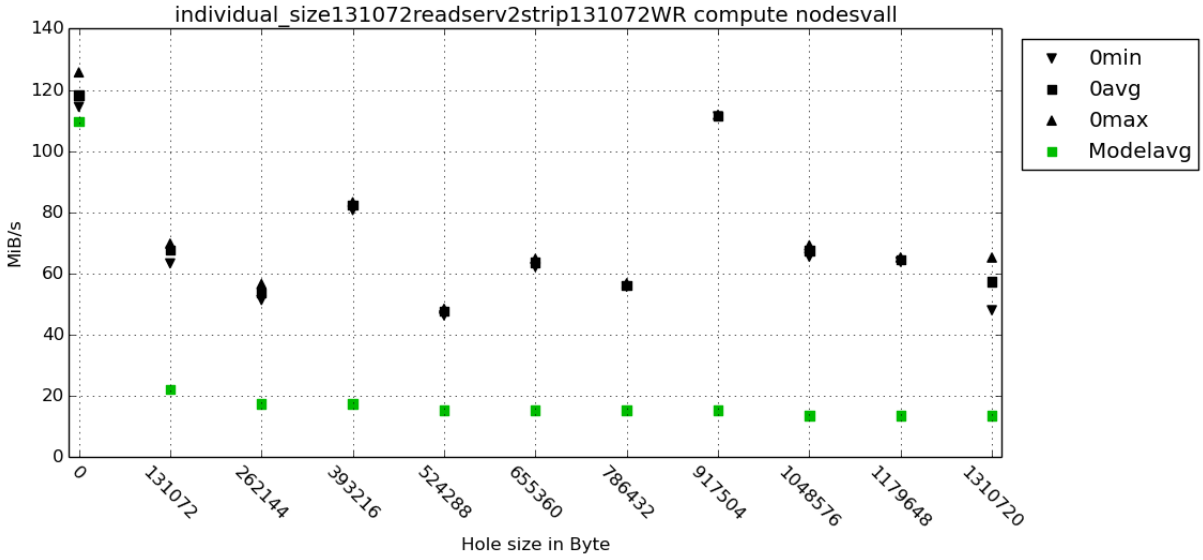

Figure 5.10.: Naive 2 node WR 640 KiByte stripe reading aligned for 124 KiByte data only

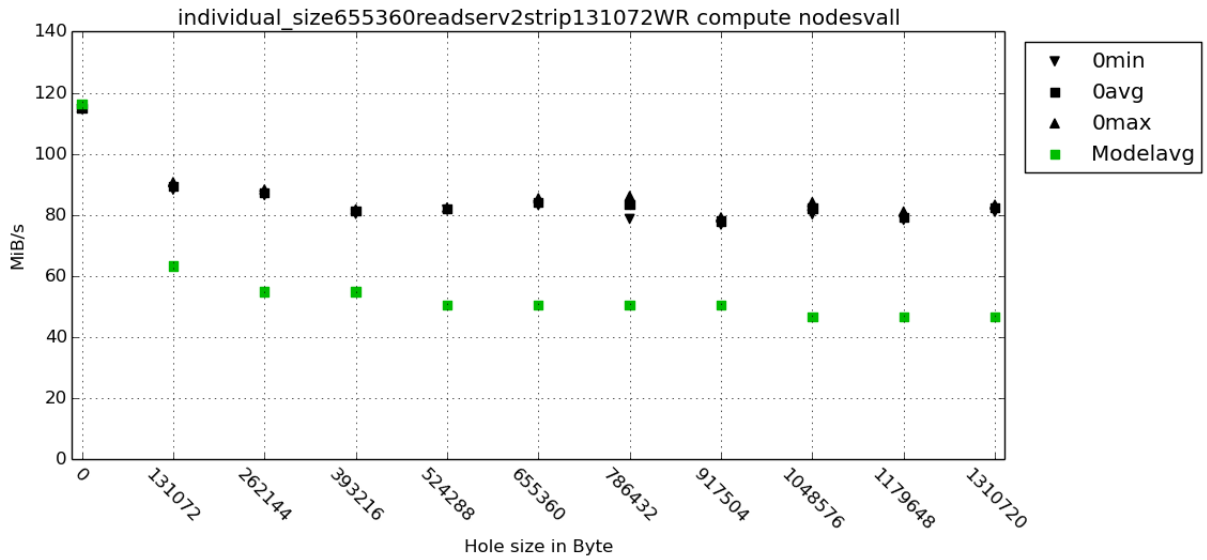

Figure 5.11.: Naive 2 node WR 1280 KiByte stripe reading aligned for 124 KiByte data only

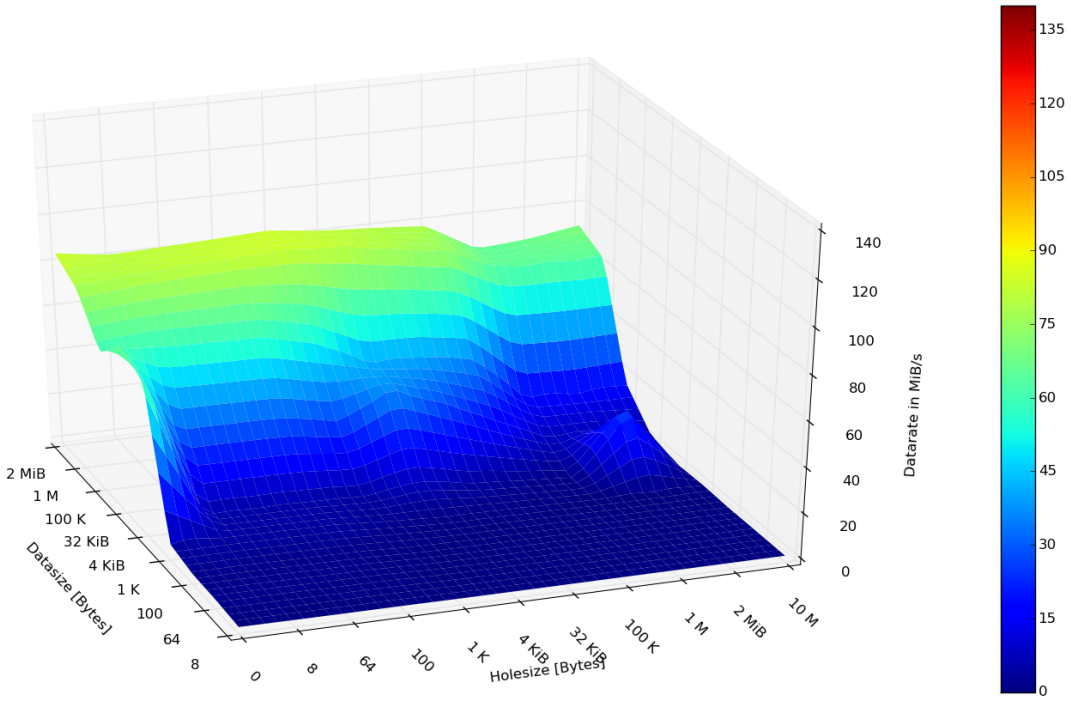

Figure 5.12.: Naive 10 node WR 128 KiByte stripe writing default pattern

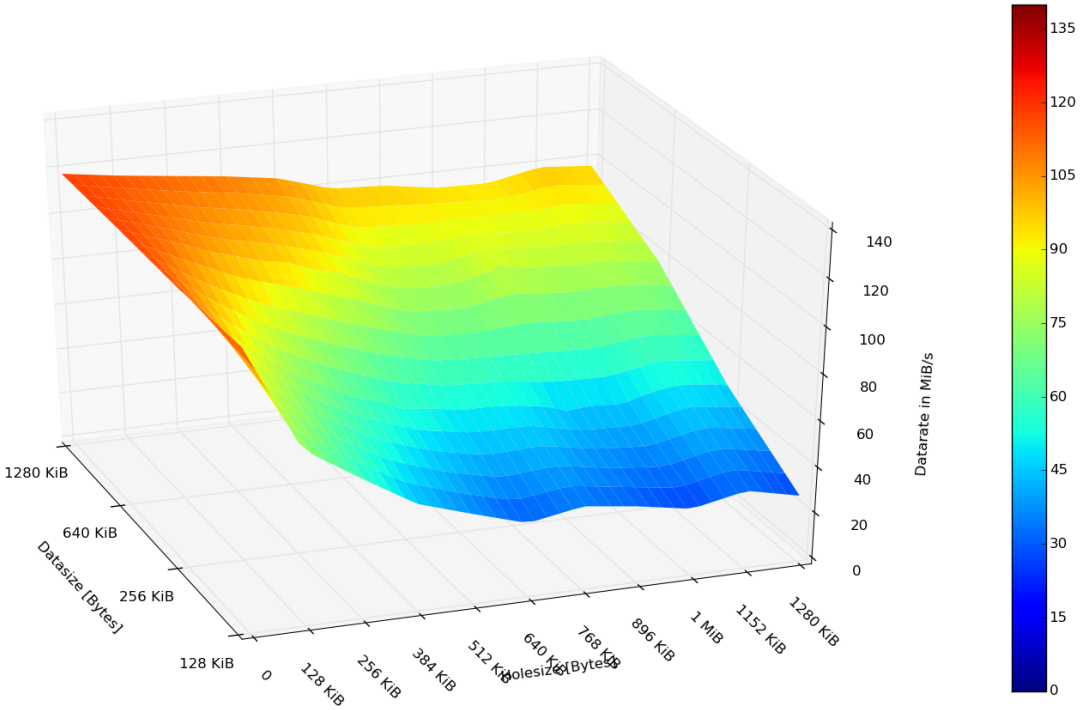

Figure 5.13.: Naive 2 node WR 128 KiByte stripe writing unaligned

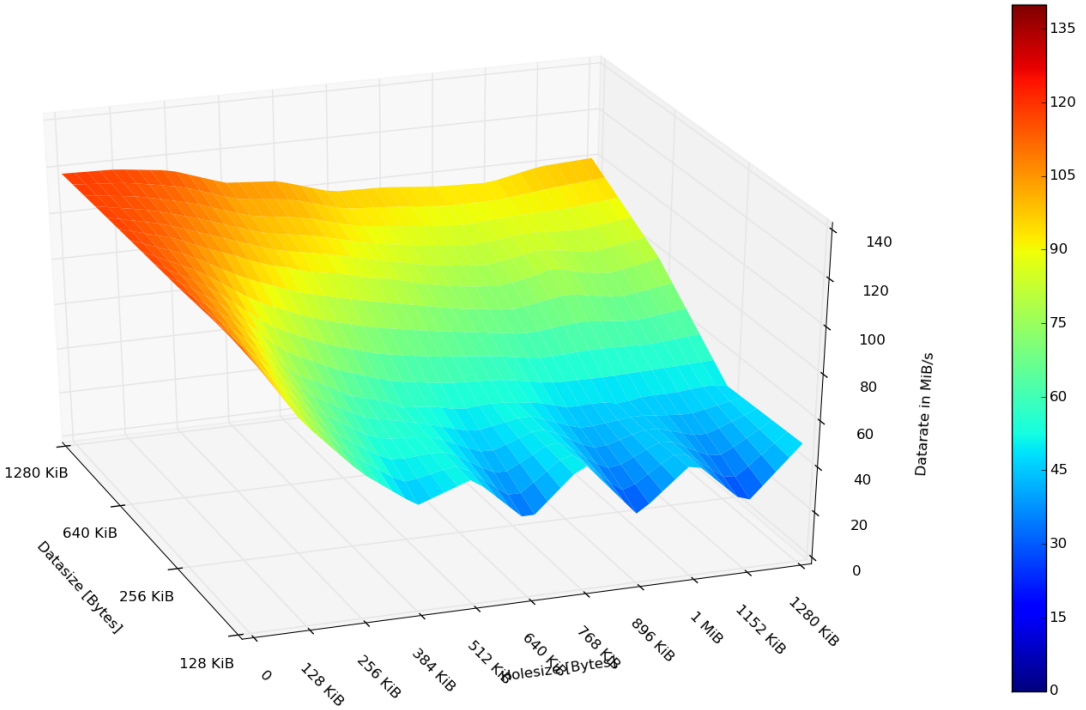

Figure 5.14.: Naive 2 node WR 128 KiByte stripe writing aligned

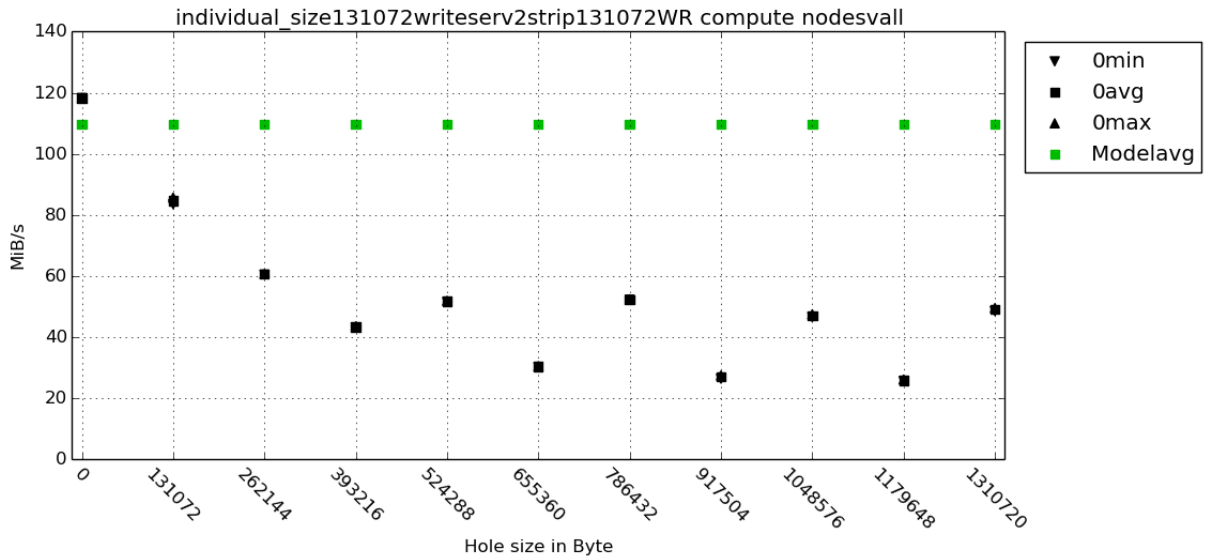

Figure 5.15.: Naive 2 node WR 128 KiByte stripe writing aligned for 124 KiByte data only

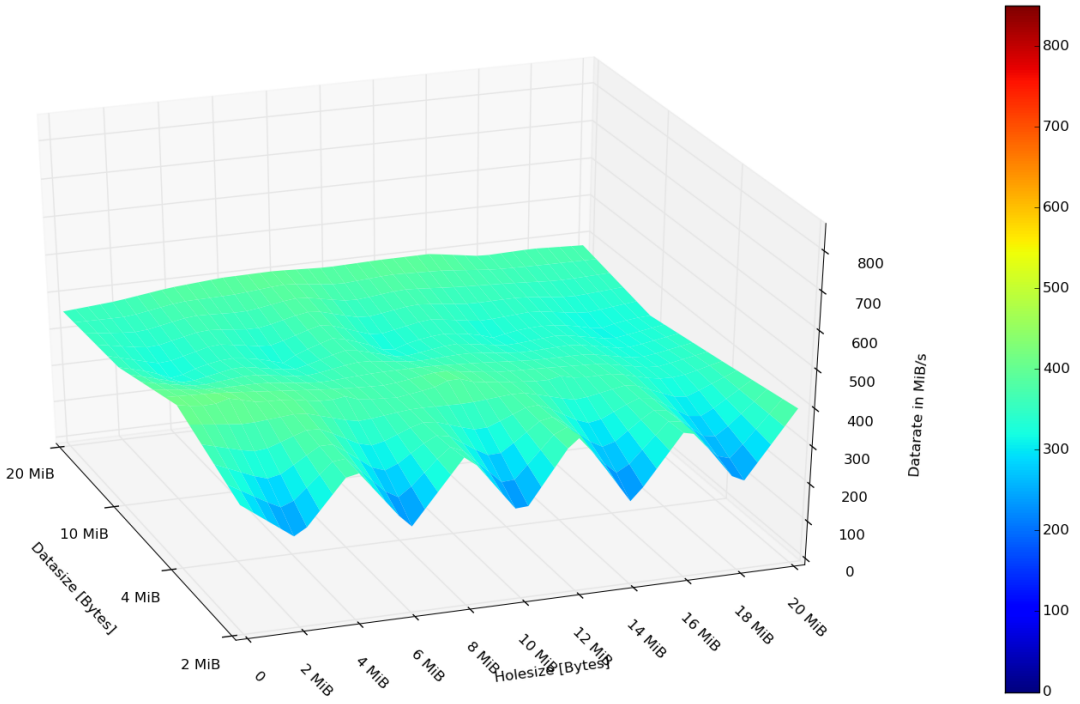

Figure 5.16.: Naive 2 node DKRZ 2 MiByte stripe writing aligned

5.3.4. Romio

This implementation always accesses as much data as fits in the DS buffer, starting at the next data block to access. So if it hits another data block due to small hole sizes it should perform well. On big holes this approach is expected to perform even worse than the naive algorithm.

As expected accessing through the data sieving buffer gives good performance as long as the hole size is smaller than 100 KiByte and the data size. For the other cases to much holes are accessed and on top it prevents other caching effects. Figure 5.17 shows ROMIO for 2 nodes with 2 MiByte stripes. So the main flaw of ROMIO data sieving is that it does not know where to stop and in this way is even harmful in some cases. This can be seen in Figure 5.18 for data sizes above 4 KiByte and hole size above 1 MByte.
In Figure 5.19 writing based on ROMIO shows the disadvantage of the read-modify-write process which is needed to write with data sieving. But it still provides a good performance increase for small hole sizes as seen in Figure 5.20. The inferior performance in writing big data sizes with read-modify-write therefore should be considered by advanced data sieving methods.

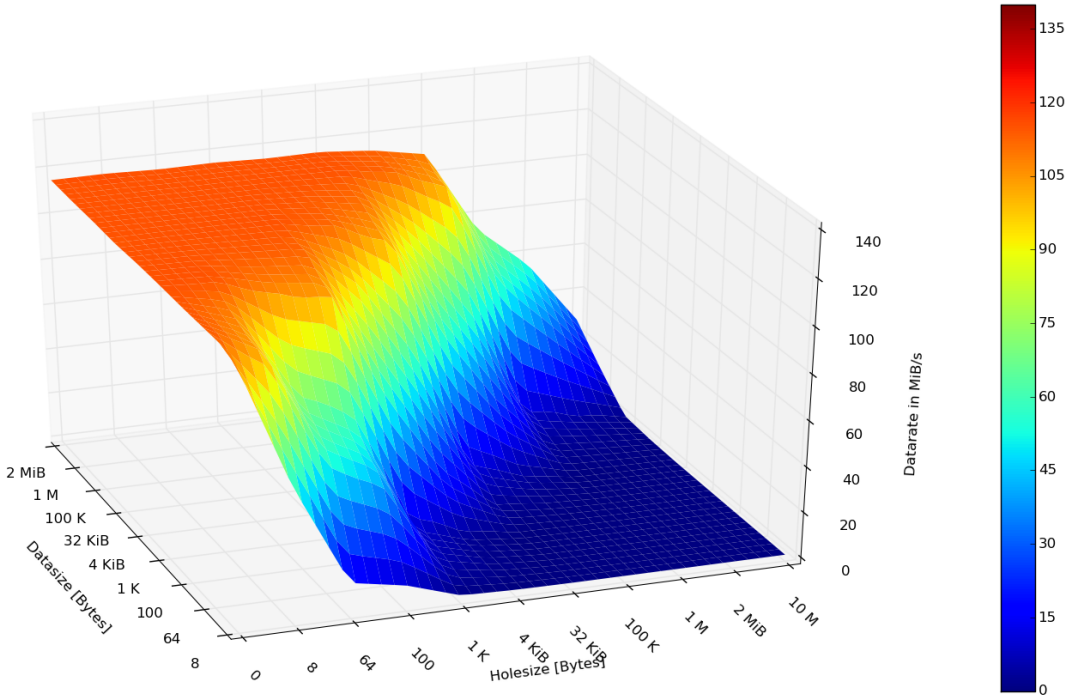

Figure 5.17.: ROMIO WR cluster read 2 nodes 2 MiByte stripes

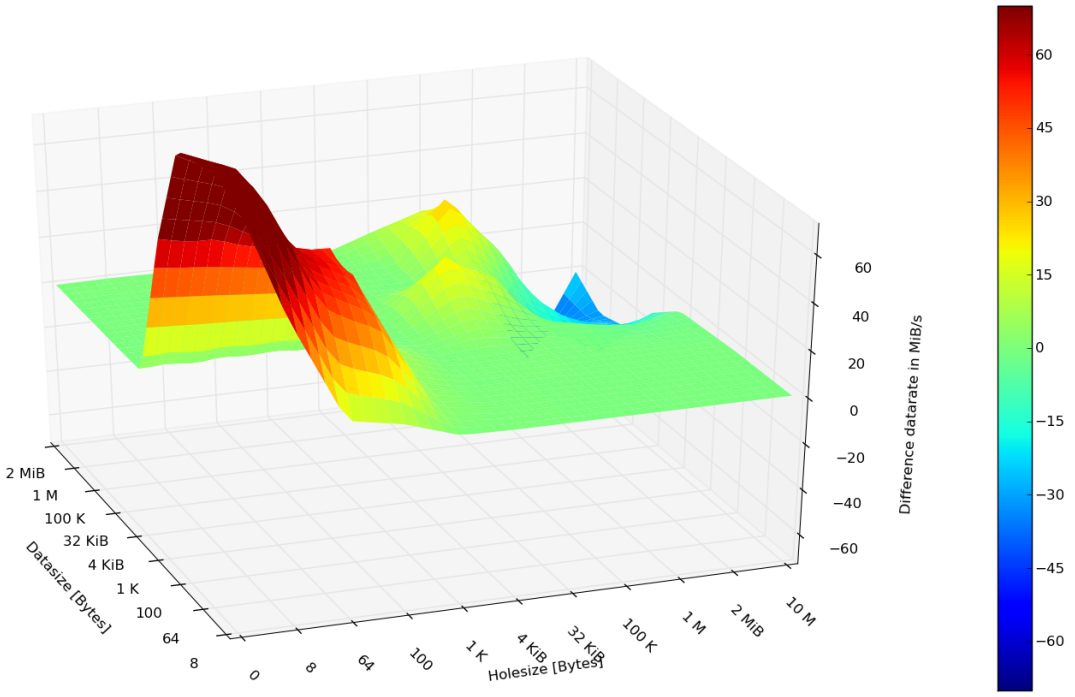

Figure 5.18.: ROMIO WR cluster write 2 nodes 2 MiByte stripes difference to naive

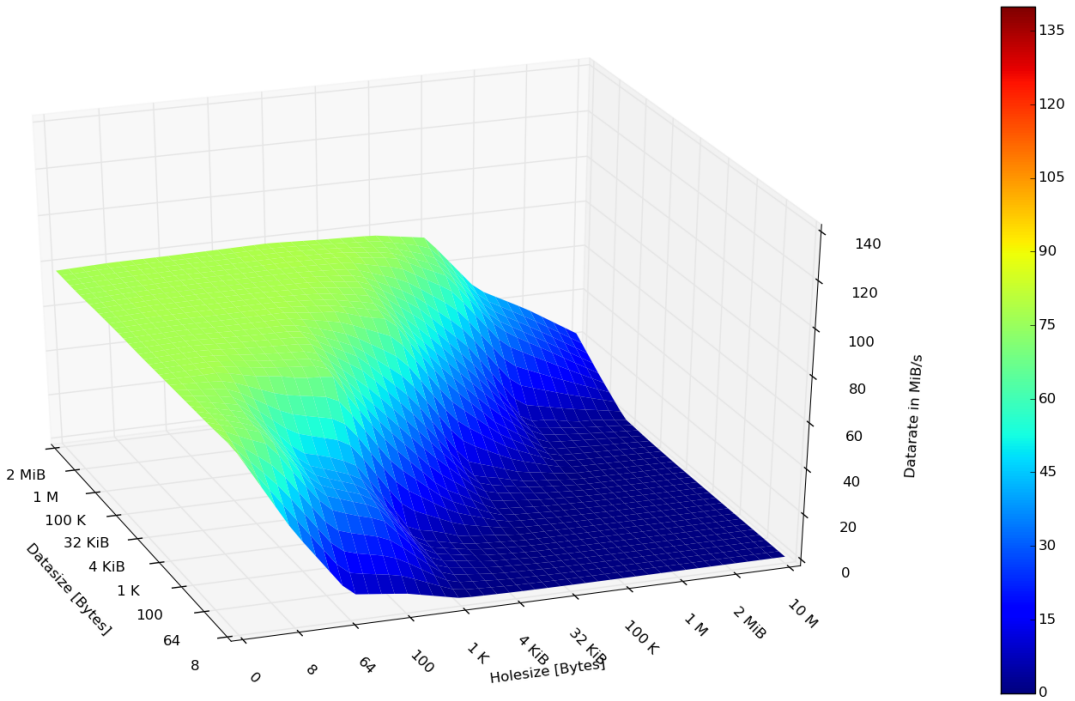

Figure 5.19.: ROMIO WR cluster read 2 nodes 2 MiByte stripes

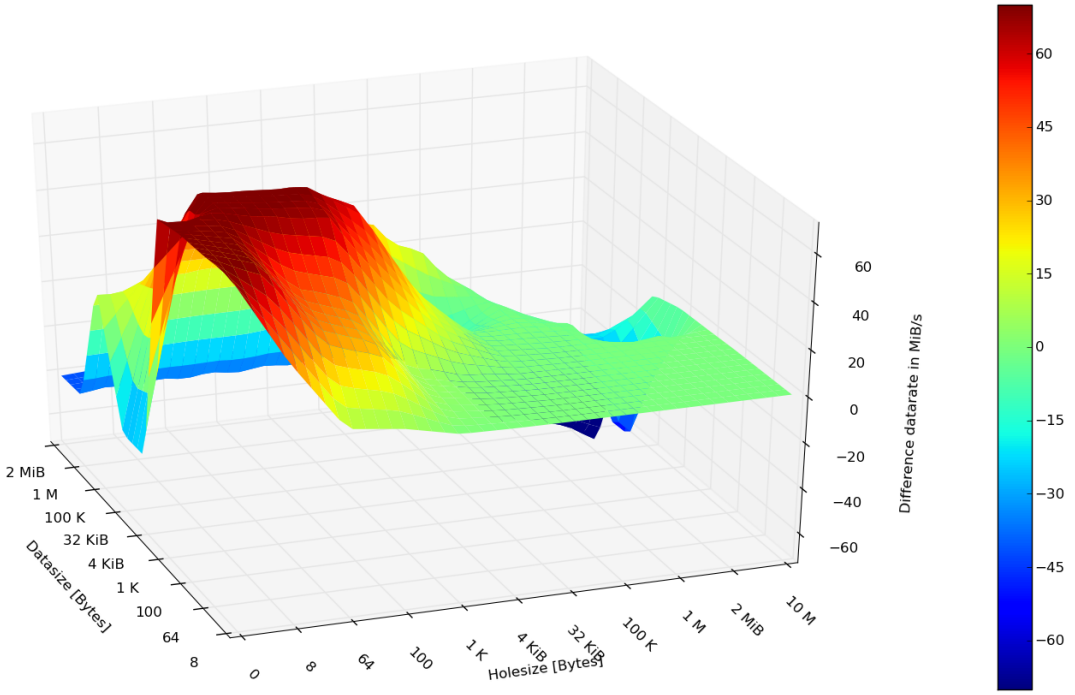

Figure 5.20.: ROMIO WR cluster write 2 nodes 2 MiByte stripes difference to naive

5.3.5. simple pm

This performance model determines dynamically if data sieving is beneficial, based on the systems performance, as closer described in sections 3.3.3 and 4.5. Figures 5.21 and 5.22 show that it improves reading data where it can or does nothing in the cases were data sieving is not beneficial. Figures 5.23 and 5.24 still show the leak of adapting to the read-modify-write overhead. For big data sizes and big holes the expectation was to measure the performance of the naive algorithm. This is not achieved due to the fact that the benchmarked version still used the read-modify-write even if only data from one tuple were accessed. It also seem to prevent caching effects as the data sieving algorithms using the *__nct_access_aggregated_tuple* lock individual for every access. The naive implementation only does it once.

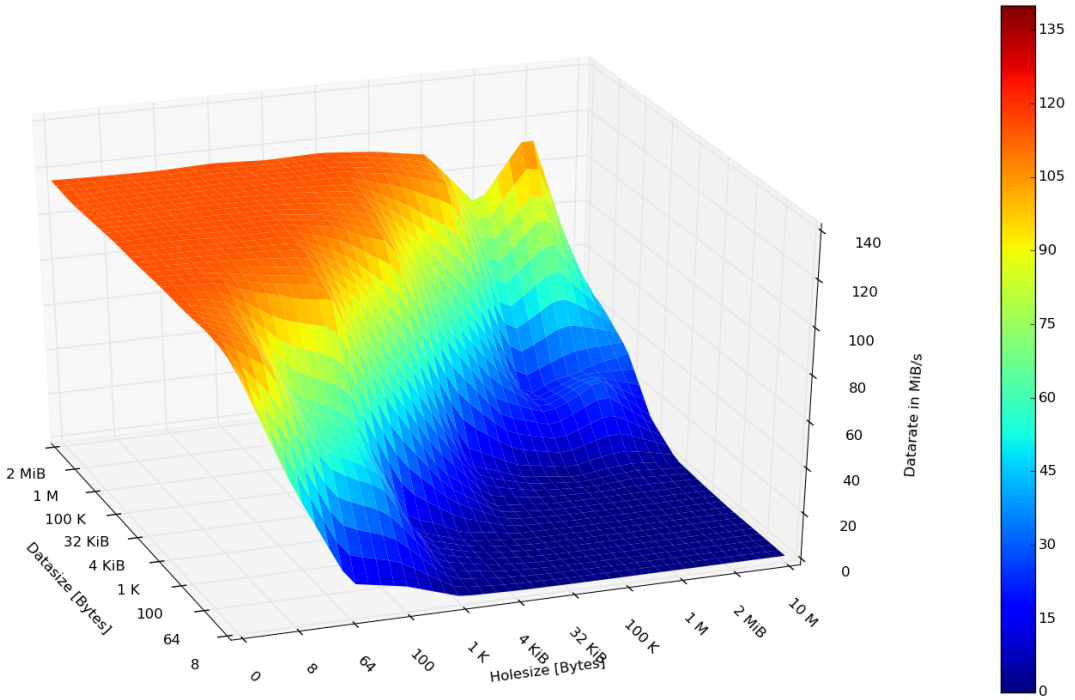

Figure 5.21.: Simple performance Model WR cluster reading 2 nodes 2 MiByte stripes

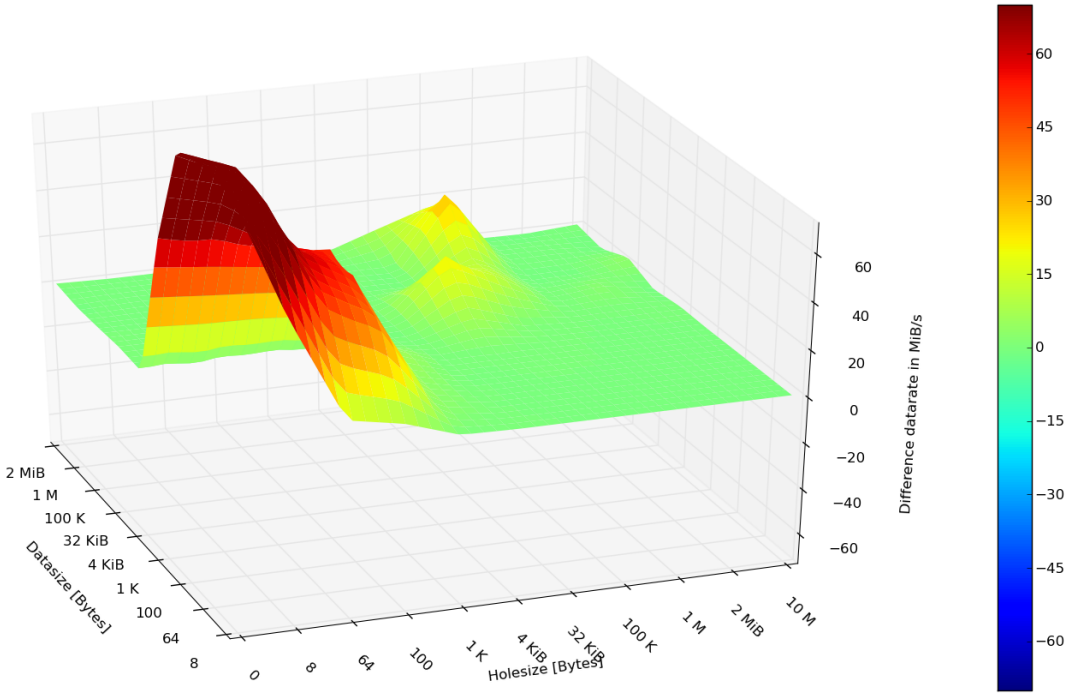

Figure 5.22.: Simple performance Model WR cluster reading 2 nodes 2 MiByte stripes difference to naive

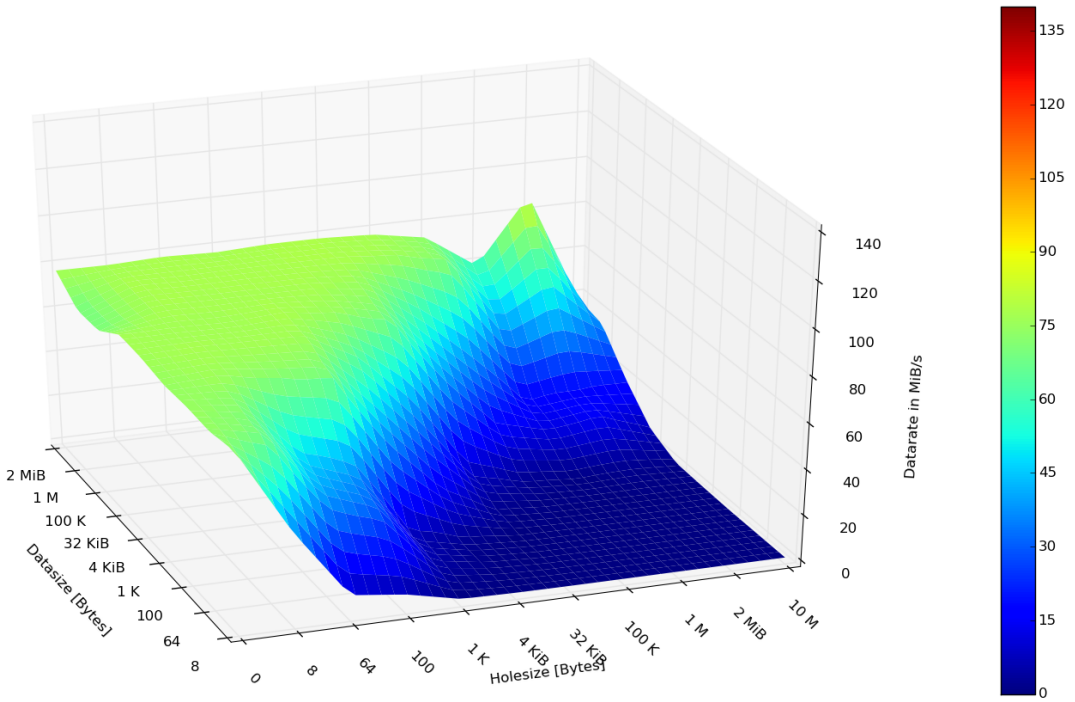

Figure 5.23.: Simple performance Model WR cluster writing 2 nodes 2 MiByte stripes

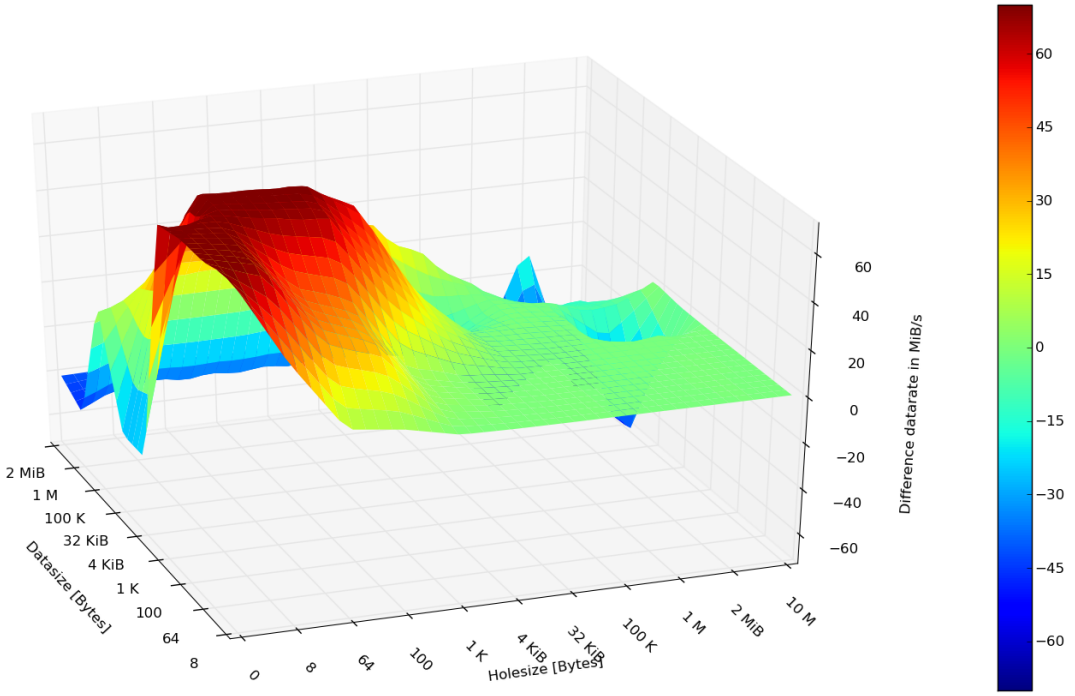

Figure 5.24.: Simple performance Model WR cluster writing 2 nodes 2 MiByte stripes difference to naive

5.3.6. Adaptive Data Sieving

Even though there was no implementation of this idea to benchmark, the previous benchmarks have shown at different points that alignment of data access matters. Therefore, the design for the advanced data sieving is proven to be a beneficial optimization to non-contiguous data access and should be further evaluated.

5.4. Conclusion

The evaluation of access methods on a great variety of data pattern proved that there is still room for improvement on current data sieving implementations. Expected flaws like the ROMIO one, of not adapting to the data pattern, were shown. Available solution like the performance model were proven to be beneficial but also still having room for further optimization, especially in the suspected area of aligning accesses and making data sieving more adaptive to the underlying system. The lack of optimization on data sieving for write access has also shown up and provides more possibility to increase performance on future data sieving algorithms. Therefore, a better understanding of current data sieving solutions is given by this Chapter and the assumption is validetad that new solutions need to be developed.

6. Summary and Future Work

This chapter contains a summary of the thesis and gives a prospect on future work.

6.1. Summary

Providing data sieving on the POSIX level by implementing a new library is a great basis to improve the capabilities of data sieving. The use of data sieving apart from MPI as well as the possibility to alter the behavior of the data sieving by a simple aggregate function, made the research on non-contiguous data access easier and more flexible. The design and implementation of the NCT library presented in Chapters 3 and 4 are therefore a great success on the goal of implementing a MPI independent data sieving library. NCT showed its advantages during the evaluation of different access methods in Chapter 5 due to its flexibility. As the evaluation has shown that further improvement on non-contiguous data access is possible, the goal of analyzing current methods to find these cases for improvements were also achieved. The results are helpful at further comprehension of non-contiguous I/O on modern cluster systems. The goal of developing a new advance data sieving algorithm were achieve just partially. Just a concept of an advanced data sieving algorithm based on results of Chapter 5 was created. The implementation and evaluation of this method were not done due to time limitations. However the expected possibility for improvements were shown to be right and provide therefore a basis for further research.

6.2. Future Work

Implementing the advance data sieving algorithm and adoption to write access will be done in future work as they both seem to provide good performance enhancement. Also some refactoring and writing a manual for NCT is needed to release it and make easy used possible. After that a integration of NCT to MPI-I/O is next. Based on that further optimization of data sieving algorithms with machine learning tools is planned.

Bibliography

[BdRC93] R. Bordawekar, J. M. del Rosario, and A. Choudhary. Design and evaluation of primitives for parallel i/o. In *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, Supercomputing '93, pages 452–461, New York, NY, USA, 1993. ACM.

[BISC08] Javier Garc Blas, Florin Isaila, David E. Singh, and J. Carretero. View-based collective i/o for MPI-IO. pages 409–416. IEEE, May 2008.

[CLA+14] Yong Chen, Yin Lu, Prathamesh Amritkar, Rajeev Thakur, and Yu Zhuang. Performance model-directed data sieving for high-performance i/o. *The Journal of Supercomputing*, pages 1–25, September 2014.

[DF12] Jack Dongarra and Message Passing Interface Forum. *MPI: a Message Passing Interface Standard: Version 3.0; Message Passing Interface Forum, September 21, 2012*. High-Performance Computing Center, 2012.

[JKH+08] Chen Jin, Scott Klasky, Stephen Hodson, Weikuan Yu, Jay Lofstead, Hasan Abbasi, Karsten Schwan, Matthew Wolf, W. Liao, Alok Choudhary, and others. Adaptive io system (adios). *Cray User's Group*, 2008.

[KMB14] Kunkel, Julian, Michaela Zimmer, and Betke, Eugen. Predicting performance of non-contiguous i/o with machine learning, 2014.

[Kun06] Julian Martin Kunkel. Performance analysis of the PVFS2 persistency layer, 2006.

[Kun13] Julian Kunkel. *Simulation of Parallel Programs on Application and System Level*. PhD thesis, Universität Hamburg, 2013.

[KZH+14] Julian M. Kunkel, Michaela Zimmer, Nathanael Hübbe, Alvaro Aguilera, Holger Mickler, Xuan Wang, Andriy Chut, Thomas Bönisch, Jakob Lüttgau, Roman Michel, and others. The SIOX architecture–coupling automatic monitoring and optimization of parallel i/o. In *Supercomputing*, pages 245–260. Springer, 2014.

[Ora11] Oracle. Lustre file system operations manual for lustre - version 2.0, January 2011.

[Sch14] Daniel Schmidtke. *Analyse und Optimierung von nicht-zusammenhängende Ein-/Ausgabe in MPI*. PhD thesis, Universität Hamburg, 2014.

[TBC+94] R. Thakur, R. Bordawekar, A Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel i/o. In *Scalable Parallel Libraries Conference, 1994., Proceedings of the 1994*, pages 119–128, October 1994.

[TGL96] Rajeev Thakur, William Gropp, and Ewing Lusk. An abstract-device interface for implementing portable parallel-i/o interfaces. In *Frontiers of Massively Parallel Computing, 1996. Proceedings Frontiers' 96., Sixth Symposium on the*, pages 180–187. IEEE, 1996.

[TGL99] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers' 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[TGL02] Rajeev Thakur, William Gropp, and Ewing Lusk. Optimizing noncontiguous accesses in MPI–IO. *Parallel Computing*, 28(1):83–105, 2002.

[TOP15] TOP500. List statistics | TOP500 supercomputer sites, January 2015.

[WR13] WR. Verfügbare hardware [scientific computing // wissenschaftliches rechnen], June 2013.

[ZJD09] Xuechen Zhang, S. Jiang, and K. Davis. Making resonance a common case: A high-performance implementation of collective i/o on parallel file systems. In *IEEE International Symposium on Parallel Distributed Processing, 2009. IPDPS 2009*, pages 1–12, May 2009.

List of Figures

List of Tables

Listingverzeichnis

Abkürzungsverzeichnis

HPC high-performance computing

I/O input and output

NAS Network Attached Storage

SAN Storage Area Network

SMB Server Message Block

NFS Network File System

SSD solid state disk

HDD hard disk drive

Appendices

A.

A.1. Job Script

```
#!/bin/bash
#
# Jobname
#SBATCH --job-name=nct_ds_spm_bench
# Dependencies just one job at
#SBATCH --dependency=singleton
# Run 10 tasks on 2 nodes.
#SBATCH -N 1 -n 1
mkdir -p ./../output
# Output goes to job.out, error messages to job.err.
#SBATCH --error=./../output/%j.out --output=./../output/%j.out

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./
mkdir -p ./results ./errors

export DS_MODE=2

#datasize of test filen in MB
fileSize=10000
#datasoieve Buff in MiB
dsBuf=4

# finishing on SIGINT and SIGTERM
trap "echo Exited!; exit;" SIGINT SIGTERM

#number of test runs
for iter in $(seq 3); do
    #number of luster servers
    for numServer in 1 2 5 10; do
        #stripe size on lustre server
        for stripeSize in 131072 2097152 ; do
            # create new folder on lustre server
            path="/mnt/lustre/zickler/nct_bench/${numServer}-${stripeSize}"
            mkdir -p "$path"
            lfs setstripe -s $stripeSize -i 0 -c $numServer $path
            #read or write
            for rwFlag in 0 1; do
                #data size in multiple from stripe size
                for dataSize in 8 64 100 1000 4096 32768 100000 1000000 2097152; do
                    #hole size in multiple from stripe size
                    for holeSize in 0 8 64 100 1000 4096 32768 100000 1000000 2097152
                        10000000; do
                        filename="/home/zickler/14-adaptive-datasieving/nct/benchmark/${
                            SLURM_JOB_NAME}/results/${numServer}_${stripeSize}_${rwFlag}
                            _${dataSize}_${holeSize}_${iter}.txt"
                        #check for existans of nonempty file
                        if [ ! -s $filename ]; then
                            tmpfile="/home/zickler/14-adaptive-datasieving/nct/benchmark/
                                ${SLURM_JOB_NAME}/results/curentRun.txt"


                            # START outut for slurm job file
                            # drop cache every benchmark
                            echo  "`date +"%x %T"`: Drop caches "
                            srun --ntasks-per-node=1 sudo /home/hr/drop-caches.sh
                            echo " "
```

```
53
54                              echo "`date +"%x %T"`: Start ${iter}. run on ${SLURM_JOB_NAME
                                    }"
55                              echo "                        Lustre: ${numServer} ${stripeSize}
                                    "
56                              echo "                        Program: ${rwFlag} \"${dataSize}-$
                                    {holeSize}\" "
57                              # END output slurm job file
58
59                              # START outut for benchmark file
60                              date > $tmpfile
61                              hostname >> $tmpfile
62                              echo "SLURM_JOB_ID: $SLURM_JOB_ID" >> $tmpfile
63                              echo "========== Lustre settings ========== " >> $tmpfile
64                              echo "Number of server: $numServer" >> $tmpfile
65                              echo "Stripesize: $stripeSize" >> $tmpfile
66                              echo "========== Run benchmark ==========" >> $tmpfile
67
68                              # Run benchmark
69                              srun ./test.exe "${path}/volume.data" "$((fileSize*1000))"
                                    $dsBuf $rwFlag 1 1 1 "${holeSize}" "${dataSize}-${
                                    holeSize}" >> $tmpfile
70
71                              # check for error on benchmark an redirect out put on error
72                              if [ $? == "0" ]; then
73                                  mv $tmpfile $filename
74                              else
75                                  errfile="/home/zickler/14-adaptive-datasieving/nct/
                                        benchmark/${SLURM_JOB_NAME}/errors/${numServer}_${
                                        stripeSize}_${rwFlag}_${dataSize}_${holeSize}_${iter}
                                        _${SLURM_JOB_ID}.txt"
76                                  mv $tmpfile $errfile
77                                  echo "`date +"%x %T"`: benchmark exited with error.
                                        output moved to error folder"
78                              fi
79                              # END outut for benchmark file
80
81                              # START outut for slurm job file
82                              echo  "`date +"%x %T"`: Finished run "
83                              echo " "
84                              # END output slurm job file
85                          fi
86
87      # finishing on SIGINT and SIGTERM
88      trap "echo Exited!; exit;" SIGINT SIGTERM
89
90                      done
91                  done
92              done
93          done
94      done
95  done
```

Listing A.1: Example of the used Jobscripts

A.2. Source Code of NCT

The source code can be found on the CD in the printed versions, otherwiese feel free to contact me.

A.3. Benchmark results

As there are too many results from the benchmarks to print them, a SQLite with the results is provied on the CD.

Erklärung

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

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

Hamburg, den 29.01.2015 .....