An in-depth analysis of parallel high level I/O interfaces using HDF5 and NetCDF-4

Master Thesis

Scientific Computing
Department of Informatics
MIN Faculty
University of Hamburg

Name:	Christopher Bartz
E-Mail:	7bartz@informatik.uni-hamburg.de
Student number:	5979025
Subject:	Computer Science
First reviewer:	Prof. Dr. Thomas Ludwig
Second reviewer:	Prof. Dr.-Ing. Stephan Olbrich

Hamburg, April 7, 2014

Abstract

Scientific applications store data in various formats. HDF5 and NetCDF-4 are data formats which are widely used in the scientific community. They are surrounded by high-level I/O interfaces which provide retrieval and manipulation of data.

The parallel execution of applications is a key factor regarding the performance. Previous evaluations have shown that high-level I/O interfaces such as NetCDF-4 and HDF5 can exhibit suboptimal I/O performance depending on the application's access patterns.

In this thesis we investigate how the parallel versions of the HDF5 and NetCDF-4 interfaces are behaving when using Lustre as underlying parallel file system. The I/O is performed in a layered manner: NetCDF-4 uses HDF5 and HDF5 uses MPI-IO which itself uses POSIX to perform the I/O. To discover inefficiencies and bottlenecks, we analyse the complete I/O path while using different access patterns and I/O configurations.

We use IOR for our analysis. IOR is a configurable benchmark that generates I/O patterns and is well known in the parallel I/O community. In this thesis we modify IOR in order to fulfil our needs for analysis purposes.

We distinguish between two general access patterns for our evaluation: disjoint and interleaved. Disjoint means that each process accesses a contiguous region in the file, whereas interleaved is an access to a non-contiguous region. The results show that neither the disjoint nor the interleaved access outperforms the other in every case. But when using the interleaved access in a certain configuration, results near the theoretical maximum are realised. We provide best practices for choosing the right I/O configuration depending on the need of application in the last chapter.

The NetCDF-4 interface does not provide the feature to align the data section to particular address boundaries. This is a significant disadvantage regarding the performance. We provide an implementation and reevaluation for this feature and observe perspicuous performance improvement.

When using NetCDF-4 or HDF5, the data can be broken into pieces called chunks which are stored in independent locations in the file. We present and evaluate an optimised implementation for determining the default chunk size in the NetCDF-4 interface. Beyond that, we reveal an error in the NetCDF-4 implementation and provide the correct solution.

Acknowledgements

I would like to thank Konstantinos Chasapis, Michael Kuhn and Petra Nerge for the great supervision of my work. I also thank Manuel Dolz for providing me with software which was useful for my thesis. Finally, I thank Prof. Dr. Thomas Ludwig and Prof Dr.-Ing. Stephan Olbrich for giving me the opportunity to realise the thesis.

Contents

1 Introduction **6**
1.1 High performance computing . . . 6
1.2 Parallel I/O . . . 8

2 Background **11**
2.1 MPI . . . 11
2.2 HDF5 . . . 13
2.3 NetCDF . . . 16
2.4 Lustre . . . 18
2.5 IOR . . . 20
2.6 VampirTrace . . . 21

3 Related work **23**

4 Evaluation **28**
4.1 Testbed specification . . . 29
4.2 Modifications to IOR . . . 30
4.3 Experimental design . . . 32
4.4 Disjoint pattern . . . 33
4.4.1 Contiguous data layout . . . 34
4.4.2 Chunked data layout . . . 37
4.4.3 Two-dimensional data layout . . . 38
4.4.4 Summary . . . 39
4.5 Interleaved pattern . . . 40
4.5.1 Contiguous data layout . . . 41
4.5.2 Chunked data layout . . . 46
4.5.3 Two-dimensional data layout . . . 47
4.5.4 Summary . . . 48

5 Enhancements **52**
5.1 Alignment . . . 52
5.2 Default chunksize using unlimited dimension . . . 53
5.3 Internal error when using unlimited dimension . . . 55

6 Conclusion **57**

Bibliography **60**

List of Figures **63**

List of Tables **65**

Listings **66**

Appendices **67**

A Experimental results **68**

1. Introduction

High performance computing (HPC) systems solve computationally intensive problems using a high degree of parallelism. To motivate the thesis, we provide an overview of the current state of the art of HPC systems in this chapter. Afterwards, we discuss the issues and problems which arise using parallel input / output (I/O) in these systems. Then we present the goal of this thesis and finally we briefly present the content of the following chapters.

1.1. High performance computing

High performance computing is an area of computing which requires extremely high processing and storage capabilities. It is used in fields that solve computationally intensive problems, like natural science or cryptography. HPC systems can be organised in a clustered manner (see Figure 1.1): Compute nodes, which consist of multiple processors, are connected via a high performance network. Each processor can execute an instance of a computer program, called process. I/O is usually performed by dedicated I/O nodes and storage systems. Message passing is one type of communication between the processes. The Message Passing Interface[1] ([SOW+95]) is a programming standard which allows programmers to write portable parallel programs using messages to exchange information between processes. More details on MPI will be discussed in Section 2.1. The network bandwidth is crucial to the performance, because many nodes communicate with each other. Multiple network technologies like Ethernet or Infiniband exist. Currently, Ethernet supplies a maximum theoretical data rate of 100 Gbit/s [2], whereas Infiniband supplies up to 300 Gbit/s ([Wik13]).

Tasks are processed by splitting up the problem into multiple pieces, so that each process solves only a part of the global problem. Afterwards, the processes communicate their local solutions via messages, if it is necessary. It is easily seen that this approach is not adequate for all problems, because a lot of problems have serial dependencies, thus full parallelization is not always possible. This problem is formalized in Amdahl's Law ([BM06]), which states that the speedup, which can be achieved by a parallelisation, is always bounded by $\frac{1}{f}$, whereas f denotes the fraction of the runtime of the program

[1]MPI
[2]Gbit/s stands for Gigabit per second, thus 10^9 bits per second

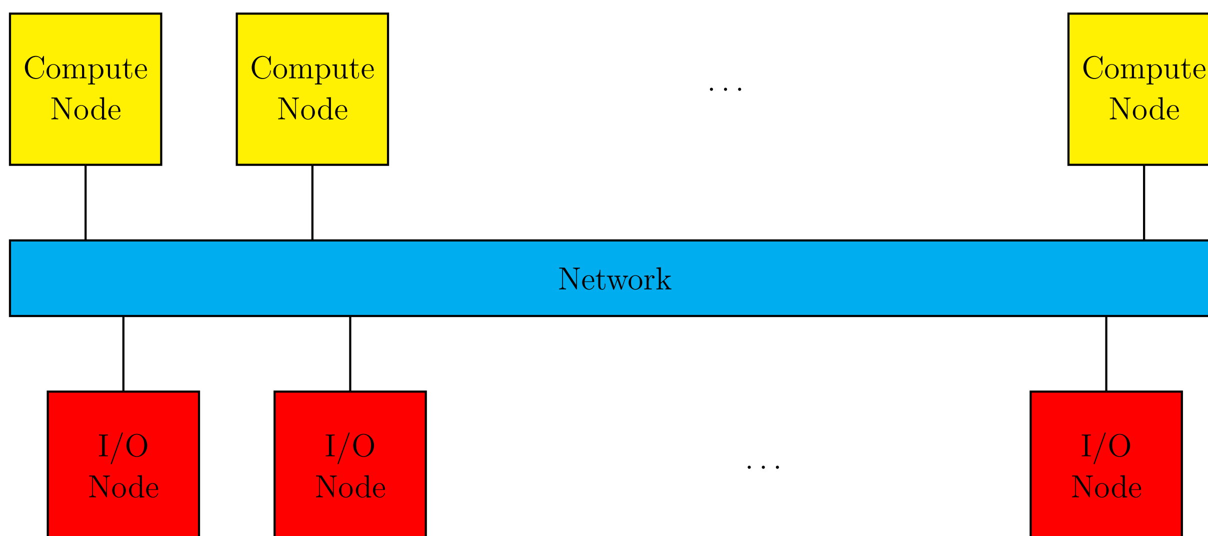

Figure 1.1.: Possible architecture of an HPC system

which has to be executed in serial. Thus, if a problem has 1% serial requirement, a maximum speedup of 100 can be achieved.

An example application using HPC is weather forecasting. As described in [BM06], the Earth's atmosphere is modeled in a discrete three-dimensional grid, each point containing various quantities like temperature and pressure. The time is also discretised, and in each timestep quantities are calculated, likely the computation considering neighbour grid points. Parallelisation makes sense in this scenario, because each process can process one or multiple points in the grid and has to communicate only with few other processes.

The computing power of supercomputers is often measured in *FLOPS*, which states the "floating point operations per second" that can be executed by the system. It is not clearly defined which kind of operation is used, but usually basic arithmetic operations like addition or multiplication are used. Furthermore, [DLP03] notes: "The performance of a computer is a complicated issue, a function of many interrelated quantities. These quantities include the application, the algorithm, the size of the problem, the high-level language, the implementation, the human level of effort used to optimize the program, the compiler's ability to optimize, the age of the compiler, the operating system, the architecture of the computer and the hardware characteristics." This means that it is not possible to have an exact specification of the amount of FLOPS a HPC system generally realises. But it is possible to define the theoretical performance of an HPC system, also called peak performance: This is the number of achieved FLOPS, when summing up the FLOPS, which each processing unit can realise (usually a value proportional to the clock cycle). Peak performance can be used as an upper bound of FLOPS which can actually be achieved when executing real world applications. To attain a more

realistic performance measure, benchmarks are used. These are standardised programs, which are executed in a controlled environment. It is important that these programs solve typical problems, which occur in most of the applications. The de-facto standard benchmark is called HPC LINPACK. As described in [DLP03], LINPACK is a collection of routines for solving various systems of linear equations, which is a common problem in multiple scientific fields. The most powerful HPC systems, as stated in the TOP500 list ([Top13]), obtain multiple PetaFLOPS using LINPACK.

1.2. Parallel I/O

High performance applications usually store data using standardised formats, so that the data can easily be exchanged and processed further. The Hierarchical Data Format 5[3]([G+00]) and Network Common Data Form 4[4] ([RDE+10]) are data formats surrounded by application interfaces (APIs), which allow manipulation or retrieval of the data by application programs written in various programming languages like C or Fortran. Both are very common in the scientific community. They store the data in a self-describing portable way using multi-dimensional arrays. More details on HDF5 and NetCDF-4 are discussed in Sections 2.2 and 2.3.

I/O performance can be crucial to overall performance of the program. For example, this can be the case if an application has to read data from disk storage before it can start processing. I/O can be performed in parallel by multiple processes, which can increase the speed dramatically in certain scenarios. The theoretical maximum of the overall read/write bandwidth equals the sum of the read/write bandwidth of each process. But the actual bandwidth can be significantly lower, because there are many factors that influence the I/O performance. For example, in most cases the disk storage will be accessed via a network, which means that the network bandwidth limits the achieveable I/O bandwidth. Furthermore, applications always use a certain access pattern when manipulating and retrieving data. The access pattern is a key factor regarding the performance. We explain this issue with a simplified example, which is illustrated in Figure 1.2. There are 16 blocks on the disk, and the application tries to access 4 blocks, which are non-contiguous. The disk has to do 4 seek operations, before it can write/read the data. If the blocks were contiguous, the disk would only be required to perform 1 seek operation. Thus, the access pattern has an impact on the performance.

Parallel I/O does also need an underlying parallel file system, because otherwise the I/O will be serialised when accessing the file system. Parallel file systems support parallel access to the same file or different files by multiple processes. The data of a file can be stored on different locations on several disks. Several parallel file systems exist, for

[3]HDF5
[4]NetCDF-4

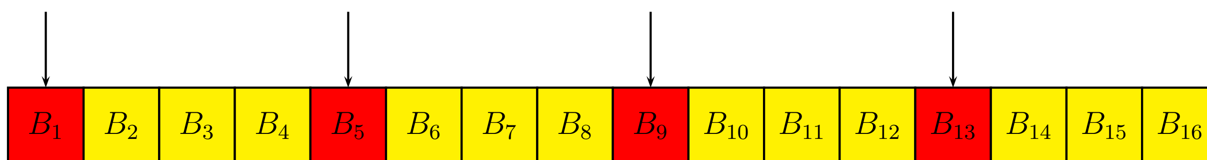

Figure 1.2.: Non-contiguous access pattern

example GPFS[5] ([BBC+98]) or Lustre ([BZ02]).

The HDF5 and NetCDF-4 APIs provide parallel access to the data. They perform the I/O in a layered manner: The NetCDF-4 API uses HDF5 to store the data, and HDF5 uses the I/O implementation of MPI which itself uses the I/O routines of the underlying system, which satisfy POSIX("Portable Operating System Interface"), a family of standards for maintaining compatibility between operating systems. Finally, I/O is performed by the I/O driver. Figure 1.3 illustrates this behaviour. If the application writes data, it uses the high-level I/O interface, and the data is passed down until it arrives at the driver layer. A read happens in the opposite direction.

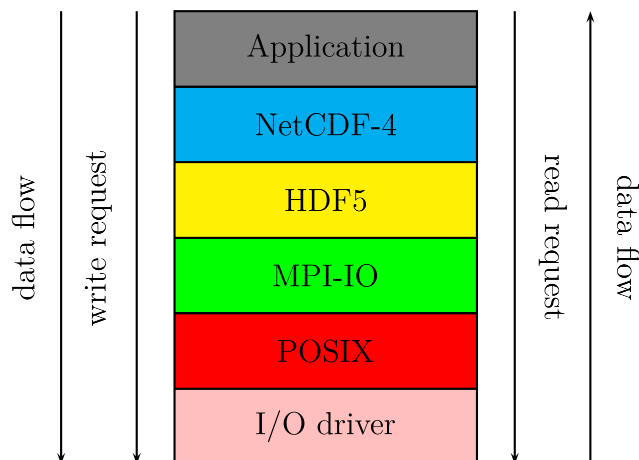

Figure 1.3.: I/O layering

The goal of this thesis is to enable high performance I/O using the aforementioned interfaces. To do this, the complete I/O path will be analysed while using different access patterns and I/O configurations. The focus is to find and eliminate the reasons for potential existing inefficiencies wherever possible and to establish I/O best practices.

To conduct our analysis we will use IOR[6] ([IORb]) benchmark. IOR is a configurable benchmark that generates I/O patterns and is well known in the parallel I/O community.

[5] General Parallel File System
[6] Interleaved Or Random

In [SS07] it is well explained why IOR is a good choice for benchmarking parallel I/O systems. The reason is that it provides many I/O configurations (like transaction size, different patterns, shared file, multiple files) and it supports the traditional POSIX interface, MPI-IO, and HDF5. In this thesis we modify IOR in order to fulfil our needs for the analysis purposes. For example, IOR lacks of the support for multi-dimensional data layout. We implement this feature.

We choose Lustre as underlying parallel file system, because it has open licensing and its architecture fits well to the hardware and network environment provided by the Scientific Computing research group at the University of Hamburg, where the evaluation takes place. Furthermore, according to [Opeb], “Currently seven of the TOP10 supercomputing sites, and over 60% of TOP100 sites, utilize Lustre.”, which means that the research results are of interest in the HPC community.

This thesis is organised as follows: Chapter 2 provides basic background knowledge, which is required to understand the subsequent chapters. We sum up work related to the analysis of parallel I/O in Chapter 3. The evaluation of the libraries using various I/O configurations and access pattern is portrayed in Chapter 4. We present enhancements to the high-level I/O libraries in Chapter 5. Chapter 6 summarises the thesis.

2. Background

In this chapter we will give an overview of various technologies, which are used in the thesis.

2.1. MPI

As discussed in the introduction, in a typical HPC architecture, the processes are decentralised and communicate using messages. The Message Passing Interface (MPI) is a standard for creating portable parallel programs using messages. It defines syntax and semantics of a set of interface functions which can be accessed from C or Fortran. This section is based on [SOW+95]. MPI has been designed from a community called MPI forum, that consist of a group of vendors, industrial users, industrial and national research laboratories and universities. Multiple implementations have been developed, for example MPICH ([MPI]) or OpenMPI ([Opea]). These implementations are available on a range of computer architectures. The standard supports efficient implementations across machines of differing characteristics by avoiding to state how operations will take place. MPI specifies only the logic of an operation and for this reason the implementation can take advantage of specific features of the underlying platform. For example, some architectures have special communication coprocessors, which means that the implementation could offload the communication to this coprocessor.

One of the key features of MPI is portability, which means that application programmers that use MPI do not have to rewrite their code for different architectures. Furthermore, MPI is also designed to allow transparency: Applications can run on systems which have subsystems of different hardware architectures. The user does not have to know how to transform the messages such that the communication between the different hardware architectures will succeed. Finally, MPI defines a minimum behaviour of all implementations, like the guarantee that the underlying transmission of messages is reliable.

An MPI program is executed by a set of autonomous processes; each process executing the code in his own environment. The processes communicate via routines which are provided by the MPI interface. Normally, each process possesses his own adress space, but shared memory scenarios are also possible. Groups are ordered sets of processes; each process possesses a unique identifier in a group which is called rank. Communicators

are either used to let a process communicate with other processes in the same group or in another group. The first type is called intracommunicator, whereas the latter type is called intercommunicator. There is also a predefined communicator called `MPI_COMM_-WORLD` which is used for communication in the group which consists of all processes.

Point-to-point communication is the basic communication type used in MPI - the transmission of data between two processes: One side sending, the other receiving. The data which is transmitted can be typed, in order to support heterogenous systems, so that the application program doesn't have to do data conversion for different architectures. Furthermore, the data can be tagged. A tag is an integer and marks the message in a special way so that the receiving part can select the messages appropiate to the purpose. The respective communication routines can be used in blocking and non-blocking mode: In the first mode, the routine returns only when it is sure that the operation is successfully performed. When using non-blocking mode, the routine returns immediately. This can be used to perform further computation during the transmittal of the data. Nevertheless it is required that the successful completion is tested with another routine call. Performing a send operation does not ensure that the receiver has received anything at all. However, there's a synchronous mode which enables rendezvous semantics between sender and receiver; the send operation is then only finished when the receiver has initiated the receipt.

In contrast to point to point communication, collective communication is the transmittal of data among all processes in a group specified by an intracommunicator object. MPI provides collective communication functions, for barrier synchronisation across all group members, global communication functions and global reduction operations. Barrier synchronisation is provided by a function called `MPI_Barrier`, which blocks the caller until all members of the group have called it. Global communication functions include broadcast, scatter and gather. Broadcast means the transmittal of a message from one sender to all members of the group. Scatter means the distribution of data from one root to all processes in the group. Each member of a group receives a separate piece of data. Gather designates the backward process. Reduction operations provide the ability to calculate results collectively. For example, this can be useful, if the sum of a set of integers shall be computed, whereas the integers are stored in a distributed manner in buffers of several processes.

The first version of the MPI standard didn't include definitions for performing parallel I/O. This feature has been added to the second version of the MPI standard ([For09]). The collection of I/O routines is also called MPI-IO. They consist of basic operations such as open/close a file or read/write from/to a file. Furthermore, more advanced features, like the creation of filetypes and fileviews are also provided. Filetypes and fileviews can be used to partition the file among processes. Moreover, MPI-IO distinguishes the I/O in two fundamental modes: independent versus collective. In the first case, the client processes neither do coordination or synchronisation when performing the I/O. In the second case, the client processes are coordinating the I/O, which provides the ability to

perform optimisations.

The most widely available MPI-IO implementation is called ROMIO ([Lab]). It is a high-performance, portable implementation which is packaged with MPICH and OpenMPI. In ROMIO there are two key optimisations, which are well explained in [TGL99]. The first, data sieving, uses the fact that multiple accesses at different locations on a disk are much more expensive than one large contiguous access (due to the overhead of the seek operations). Therefore, multiple independent accesses are aggregated into one large access. By doing this, data regions are accessed which should not be accessed. These data regions are filtered out when the whole data has arrived at the client in the application buffer. The second key optimisation is called Two-Phase I/O. It is an optimisation which is applied when the I/O is performed collectively. It takes advantage from the knowledge of the entire access information of a group of processes. Similar to data sieving it is based on the assumption, that large accesses are better than small independent accesses. The algorithm figures out if the accesses of the processes are interleaved. In such cases, the data accesses are reorganised, and the I/O is delegated to I/O aggregator nodes (which can be a subset of all client nodes), in the manner that each node has access to a separate contiguous region. Later, when the data is transferred to the client I/O nodes, the data is reorganised, and each client node accesses the data regions it has originally requested. The Two-Phase I/O is illustrated in Figure 2.1.

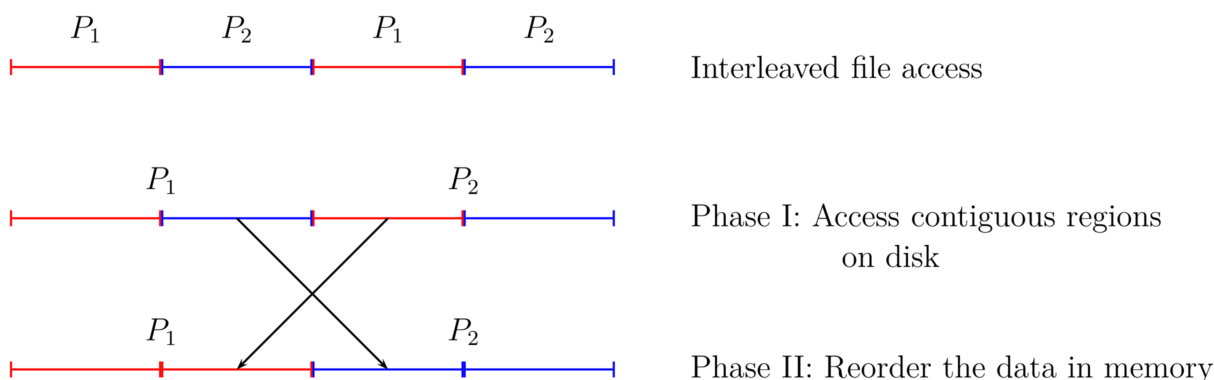

Figure 2.1.: Two-Phase I/O

2.2. HDF5

HDF5 is a self-describing portable data format, surrounded by a high-level I/O interface which provides retrieval and manipulation of data. This section is based on the content of [Kor11] and [G$^+$00]. Implementations for a range of platforms exist and high-level

APIs for various programming languages like C, C++, Fortran 90 and Java are provided. HDF5 supports large and heterogenous data: There is no limit on the number or size of data objects and it is possible to mix any kind of data within a HDF5 file. One can specify complex data types, relations and dependecies. Fast access to the data is provided through accessing parts of the file without parsing the entire content. Furthermore, many open-source and commercial tools exist for managing, manipulating and analysing the data.

HDF5 uses two basic concepts to store the data in a hierachical manner: groups and datasets. Groups contain zero or more groups or datasets; that corresponds to the concept of directories in filesystems. Datasets store the actual data in multidimensional arrays; that corresponds to the concept of files in filesystems. A dataset consists of two parts: A header and the data array. The header contains the metadata of the data which is actually stored in the array. The header saves the name of the dataset, the datatype, dataspace and storage layout of the data. Each data element has the same datatype and the type can be basic (like integer, float,...) or compounded. The dataspace is a description of the shape of the array. It consists of the number of dimensions and the current and maximum sizes of each dimension. The dataset can be extended up to the maximum size in each dimension via a routine which must be called using collective I/O. A dimension's maximum size can also be specified as unlimited, meaning that the dataset can grow without limit in this dimension. Groups and datasets can also have metadata and attributes, which are user-defined HDF5 structures containing extra information about the HDF5 object.

The data can be stored in three different ways: contiguous, compact or chunked. Contiguous means that the whole data of a dataset is stored in one contiguous array of bytes apart from the header, as illustrated in Figure 2.2. n-dimensional datasets are seralised into a 1-dimensional datastream by mapping the memory buffer directly to the disk. When using the C interface, the data is stored in a row-major order. This layout requires that the dataset must be of fixed-size (this means that it is not extendible), because it is not assured that free space is available after the array. If the data is stored

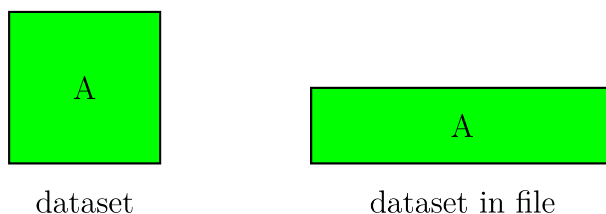

Figure 2.2.: Contiguous dataset

in a compact way, it is directly stored in the dataset header. This makes sense when the data amount is small, because in contrast to the contiguous layout, where the data is stored apart from the header, only one I/O operation is required to access the data.

Just like the contiguous data layout, the dataset must be of fixed-size. The data can also be stored using chunks, which means that the dataset is split into chunks, which are rectangular regions, and the chunks are written into independent locations in the file (illustrated in Figure 2.3). A chunk may have any size or shape that fits in the dataspace;

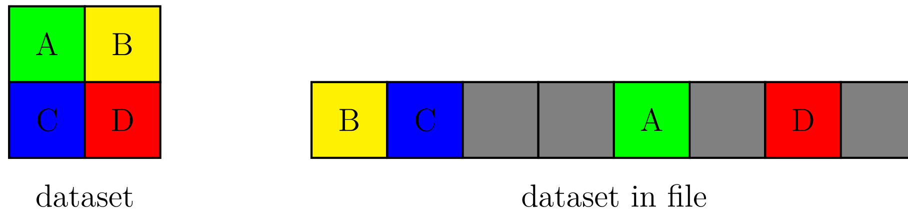

Figure 2.3.: Chunked dataset

a 3-dimensional dataspace can be split into 3, 2 or 1-dimensional chunks. Chunks may also extend beyond the size of the dataspace; the extra regions will not be accessible. For example, a 3×3 dataspace can be overlaid by 2×2 chunks, leaving 7 elements unused (illustrated in Figure 2.4). The locations of the chunks are stored in the header section of the dataset, using a tree data structure called B-tree ([BM02]). B-trees hold the stored data in sorted way and allow access and manipulation in logarithmic time. They are commonly used in databases and filesystems. The HDF5 interface provides a routine which aligns the address of the file objects (dataset header, contiguous datablock, chunks) to a multiple of the alignment value passed. This can be useful in combination with parallel I/O, which we will explore in this Thesis.

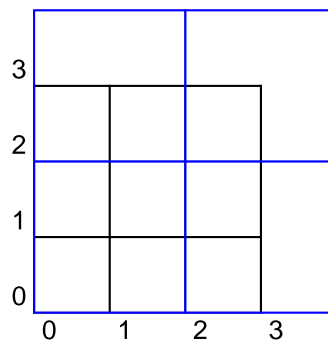

Figure 2.4.: 2×2 chunks overlaying a 3×3 dataspace

Chunks are read/written individually, and can be used to improve the performance when performing partial I/O (on a subset of the dataset), which is explained in [Gro13]. Also, the interface provides a tunable chunk cache: Each time a chunk is accessed, the whole chunk is loaded in the cache and the selected points are loaded in the user buffer. Each dataset possesses its own cache. The cache can be turned off, which is recommended for some scenarios (for example if the full dataset is queried). Furthermore, the HDF5 interface provides also the feature to (de)compress the data that it (reads)writes. Compression requires chunked layout because it is applied on chunks of the data.

HDF5 implements several strategies to allocate storage space and fill the space with fill values. Fill values enable a reading process to discover if the region has already been accessed by a writing process or not. According to the POSIX specification, “read() shall return bytes with value 0” for unaccessed regions. But if 0 values are also used by writing processes, a reading process could not decide if the data is dirty or not. If the application uses a fill value, e.g. -1, and ensures that no writing process uses this value, the region would consist of bytes with value -1 and it could be decided that this region has not been touched by another process. HDF5 supports early, late and incremental file space allocation. Early means that the space of the whole dataset is allocated immediately when the dataset is created. The whole space is allocated first when the first data is written, when using late as strategy. Incremental is only supported when using chunked layout, and in this mode only the storage for each chunk is allocated when it has to be written. When using parallel I/O, the default allocation strategy is early, as each process requires a view of the entire dataset. The filling of the space is done when the space is allocated, but can be disabled in order to improve performance. This is strongly recommended when it is clear that the full dataset is written before it is read, because the write performance degrades approximately to the half of the original write performance, as each write is performed two times.

2.3. NetCDF

NetCDF is a self-describing portable data format and interface for accessing and manipulating the data. The development is supervised and hosted by the Unidata program at the University Corporation for Atmospheric Research (UCAR). NetCDF is used in the scientific community, especially in climatology, meteorology and oceanography applications. The content of this section is based on [RDE+10]. As in HDF5, the data is stored in arrays, which are multidimensional fields; each storing a data element of a datatype, which is fixed for the whole array. The NetCDF interface provides an abstraction from the underlying storage format in the manner of an abstract datatype. The user retrieves and manipulates a NetCDF file using the software libraries, the internal representation of the data is hidden from the user, allowing portable and exchangeable implementations. The physical representation of the data is designed to be machine-independent. NetCDF supports access from various programming languages, for example C, FORTRAN 77, FORTRAN 90, and C++. Furthermore, there exist implementations for various UNIX systems; an MS Windows port is also available.

NetCDF comes in different versions. The current version is NetCDF-4. It introduced a new binary format, too. There exist three different binary formats for NetCDF files:

- classic format; this was used in the first version of NetCDF
- 64-bit format, this is an extension of the classic format, allowing larger variables

and file sizes

- NetCDF-4; this is the HDF5 data format, with some restrictions

This thesis focuses on the current version, NetCDF-4.

Generic utility programs exist, which have the purpose to visualise or modify the data. *ncdump* for example, prints the data structure in a human-readable format onto the console. *ncgen* is the counterpart of *ncdump*; it reads a description of a NetCDF file in a human-readable format, and generates a binary version of it. The textual description is in a tiny language called CDL (network Common data form Description Language). Furthermore, a set of standalone, command-line programs exists, called NetCDF Operators ([NCO]), which take NetCDF files as input, operate on them, and output the results in text, binary or NetCDF formats. Example operations are deriving new data, averaging or printing. The purpose is to aid the manipulation and analysis of gridded scientific data.

The data model of NetCDF consists of dimensions, variables and attributes. A NetCDF dimension consists of a name and a length, which is an abitrary positive integer. In the classic or 64-bit format, at most one dimension can have unlimited length; NetCDF-4 files can have multiple dimensions with unlimited length. A variable is a multi-dimensional array of values of the same type. It consists of a name, datatype and a list of dimensions, which define its shape. If a variable has at least one unlimited dimension, it is called record variable, and the possible amount of entries is unbounded. In NetCDF-4, record variables use extendible HDF5 datasets to store the data. As mentioned in Section 2.2, collective I/O is required in this case. The list of dimensions can also be empty, meaning that the variable can store only one value. When a variable has 0 dimensions, it is called scalar, with 1 dimension vector and with 2 dimensions matrix.

There exists a special type of variable, called coordinate variable. A coordinate variable corresponds to a dimension, it has the same name as the respective dimension, is a vector and its shape is the same as the dimension. The values of the variable are typically physical coordinates which are describing the values of the dimension. It has no meaning to the NetCDF interface, but can be used by application programs to specify positions along it. For example, consider there exists a dimension with size 5 called *time*, which describes the time of a measurement. A corresponding coordinate variable would have the name *time*, datatype integer and the dimension *time*. Its values could be $0, 10, 20, 30, 40$. Furthermore, assume in the dataset exists a variable called temperature with the dimension time. Each point describes the temperature which was measured at a respective time. An application program which visualizes the data could now use the values of the coordinate variable instead of just index positions $(0, 1, 2, 3, 4)$.

NetCDF distinguishes between internal and external data type of a variable. The internal type is the type used in the respective programming language in the application program. The external data type is the type used in the NetCDF file. The data is

converted when reading or writing to the file, if the physical representation is differing. External types provide portability, because the application programmer does not need to take care of the underlying storage architecture. Furthermore, the data format can be changed independently from the application programs, because the conversion is performed by the software libraries.

Attributes are used for storing metadata. They can be used globally, which means storing information about a whole NetCDF dataset. But the most common use case are variable-specific attributes. In NetCDF-4, attributes can also be placed at group level. There exist reserved attributes, which are used by the NetCDF interface, and attribute conventions for generic application software. For example, most generic applications which process NetCDF datasets use variable-specific attributes called *valid_min* or *valid_max* to specify the range of allowed values for a variable. Furthermore, a global attribute "Conventions" can be used to specify the name of a set of well defined conventions, followed by the dataset. The type of the attribute is a char array, which consist of the name of conventions. For example there exist the Climate and Forecast (CF) conventions, used to describe atmospheric, ocean and climate data.

As mentioned above, NetCDF-4 uses the HDF5 file format. By default, for non-record variables, a contiguous data layout is used. But the user can specify chunked data layout and chunk sizes for each dimension via a function. If record-variables are used, the data layout is set to chunked due to the requirement of extendibility. NetCDF-4 does also support compression of data, which requires chunked layout, too. Finally, NetCDF-4 does also provide a routine to disable the prefilling of data.

2.4. Lustre

Lustre is an open-source distributed parallel filesystem, whose deployments are popular in scientific supercomputing. It presents a POSIX interface to its clients with parallel access capabilities to the shared file objects. Our descriptions are based on [OI14]. Lustre stores data in an object-based manner: The file is split up into multiple objects ("stripes") which are stored on different servers. The distribution on different servers enables parallel I/O performance. The bandwidth is bounded by the sum of bandwith of each I/O server. The bandwith of an I/O server is bounded by the network and disk access data rate. The total storage capacity is the sum of the storage capacity of each storage server.

Lustre's architecture (see Figure 2.5, taken from [OI14]) is divided in three parts: Clients, metadata servers and storage servers. Clients are connected via network with the Lustre services, which are responsible for storing the metadata and the objects. Metadata servers (MDSes) store namespace metadata, such as filenames, directories, access permissions, and file layout on one or more metadata targets (MDTs). Lustre uses multiple

object storage servers (OSSes) to store the file data on one or more object storage targets (OSTs). Clients access and concurrently use data through the standard POSIX I/O system calls.

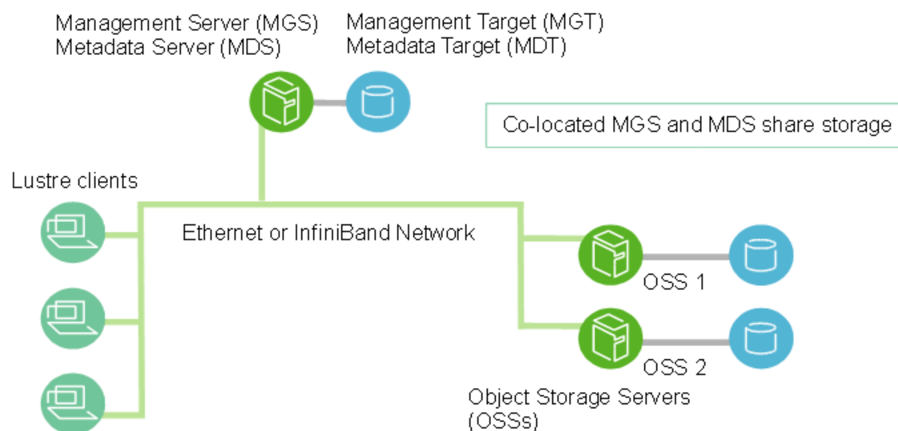

Figure 2.5.: Architecture of Lustre

File striping across multiple OSTs can be used to improve the performance. This is the case, when the aggregated bandwidth to a single file exceeds the bandwidth of a single OST. The size of a stripe is specified by a parameter called stripe size. The number of OSTs which shall be used is defined by stripe count. When the chunk of data being written to an OST exceeds stripe size, the next chunk of data in the file is stored on the next OST. The next OST is determined in a round robin fashion. In the default configuration, stripe size is set to 1 MiB and stripe count to 1, which implicates no striping across multiple servers. However these parameters can be configured by the user on a per-file or per-directory basis.

When accessing a file, each client node is able to determine with which OST he has to communicate, because the stripes are allocated in a round robin fashion. This implicates that there is no need for communication with the metadata server after the file layout is clear. Our analysis focusses only on the performance when writing/reading to/from one file, so the handling of the metadata should be negligible.

In order to protect the integrity of each file's data and metadata, Lustre possesses a distributed lock manager. Furthermore, locking enables client side caching, since other clients do not have the ability to manipulate the data while the lock is held. File data locks are directly managed by the OSTs containing the file region. The locks are provided as byte-range extent locks. Metadata locks are managed by the MDT that stores the inode for the file.

2.5. IOR

IOR is a software used for benchmarking parallel filesystems using different I/O APIs and various access patterns. The main purpose of IOR is to measure the read/write performance when accessing one or multiple files in the filesystem. IOR differentiates the strategies to use a shared file or one file per process. These strategies can be easily compared for an identical set of testing parameters. According to [SS07], "most of the existing benchmarks are not reflective of the current I/O practice of HPC applications, either because the access pattern did not correspond to that of the HPC applications, because they only exercise POSIX APIS (eg. no MPI-IO, HDF5, NetCDF), or because they measure only serial performance." In [SAS08], IOR is used to predict the performance of three I/O benchmarks, whose I/O patterns are "representative of many I/O intensive codes in the NERSC[1] workload". In this study IOR predicts the performance with a maximum error of 10 %, showing the powerfulness of IOR.

When using shared file strategy, the organisation is as follows (illustrated in Figure 2.6): The file comprises segments, which themselves comprises blocks. A block is a coherent collection of data accessed by one process. A process reads/writes to a block in chunks which have a configurable size, called transfer size. This is the amount of data which is written/read per function call in the application program. When using a one file per processor strategy, each process stores its blocks in its own file. Segments and blocks can be used to simulate different access patterns. For example, when using only one segment in a shared file, a disjoint access pattern can be simulated, because there is no interweavement between the access of the processes. When using multiple blocks, the access of the processes is interleaved. When using POSIX or MPI-IO, the data which

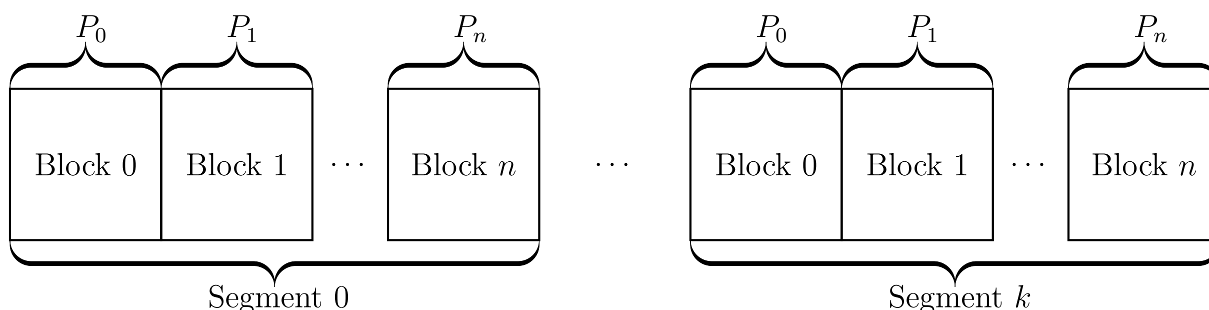

Figure 2.6.: IOR shared file

is saved to the file is just a collection of bytes, without any additional structure. IOR uses a one-dimensional dataset when HDF5 is the desired API.

IOR can be parametrised using various configurations. The following parameters of IOR

[1] National Energy Research Supercomputing Center

are important to this thesis: *api, segmentCount, blockSize, readFile, writeFile, reorderTasksConstant, transferSize, numTasks, repetitions, memoryPerNode, interTestDelay, collective, noFill, setAlignment*. *api* describes which I/O API to use. IOR version 3.0.1 (which will be used in this thesis) supports POSIX, MPI-IO, HDF5 and PNETCDF ([PNE]). The latter is a parallel I/O interface for NetCDF versions lesser than NetCDF-4. *segmentCount* indicates the count of segments to use and *blockSize* the size of each block. If *readFile* (*writeFile*) is set to 1, a read (write) operation will be measured. It is also possible that both read and write will be measured; in this case the write will be performed first and the read afterwards. This scenario has conflict potential: If the same node reads the data back which has written it, the cache will deliver high performance. For this purpose, the parameter *reorderTasksConstant* can be used. When set to x, the process with the rank $n + x$ will read back the data, which the process with rank n has written. This prevents the data from being read directly from the cache. The *transferSize* is the amount of data which each process writes/reads per function call in the application (see Chapter 1). *numTasks* indicates how much client processes will participate in the I/O operation. With *repetitions* the count of repetitions can be choosen. IOR has a built-in statistic system, such that mean values and standard deviations of the performance will be calculated. *memoryPerNode* is the amount of memory which the application will allocate to simulate application behaviour. It can also be used to prevent client-side caching, because if almost no memory is available, then the data cannot be cached in memory. *interTestDelay* is the count of seconds which will be paused between two tests. If *collective* is set to 1, then the I/O will be performed collectively (see Section 2.1). The *noFill* option indicates if the HDF5 (and NetCDF) interface will use the filling behaviour (see Section 2.2). *setAlignment* can be set to the aligning borders as explained in Section 2.2.

2.6. VampirTrace

When analysing or developing high performance programs, it is often important to discover potential bottlenecks and performance issues. VampirTrace is a tool, which can be used to analyse the behaviour of a parallel program. This section is based on [KBD+08]. We distinguish two types of program analysing methods: profiling and tracing. Profiling aggregates the information of certain event types, whereas tracing records information of individual events. Tracing is more powerful than profiling, because one can use the information gained from tracing to profile a program, but not the other way around. However, tracing produces more data and can slow down the performance of the program, which can manipulate the results. Common event types which can be recorded are

- enter or leave of function calls
- point to point communication using MPI

- collective communication using MPI
- performance counter samples

The modification of the application in order to detect event occurrences is called instrumentation. VampirTrace is an instrumentation and measurement component, which can be used to trace parallel applications. It is integrated in the current versions of Open-MPI, an implementation for the MPI standard. VampirTrace adds code to the application by compiler, linking or source-to-source instrumentation. An instrumented application generates trace files, which can be opened for visualisation by a tool called Vampir. It provides zooming and browsing in timeline and statistical displays. For example, the global timeline shows processes and threads on a vertical axis. The program state is visualised by horizontal bars, coloured according to the function which is active at this given point of time. Point to point communication, global communication and I/O operations are displayed by arrows. A screenshot can be seen in Figure 2.7.

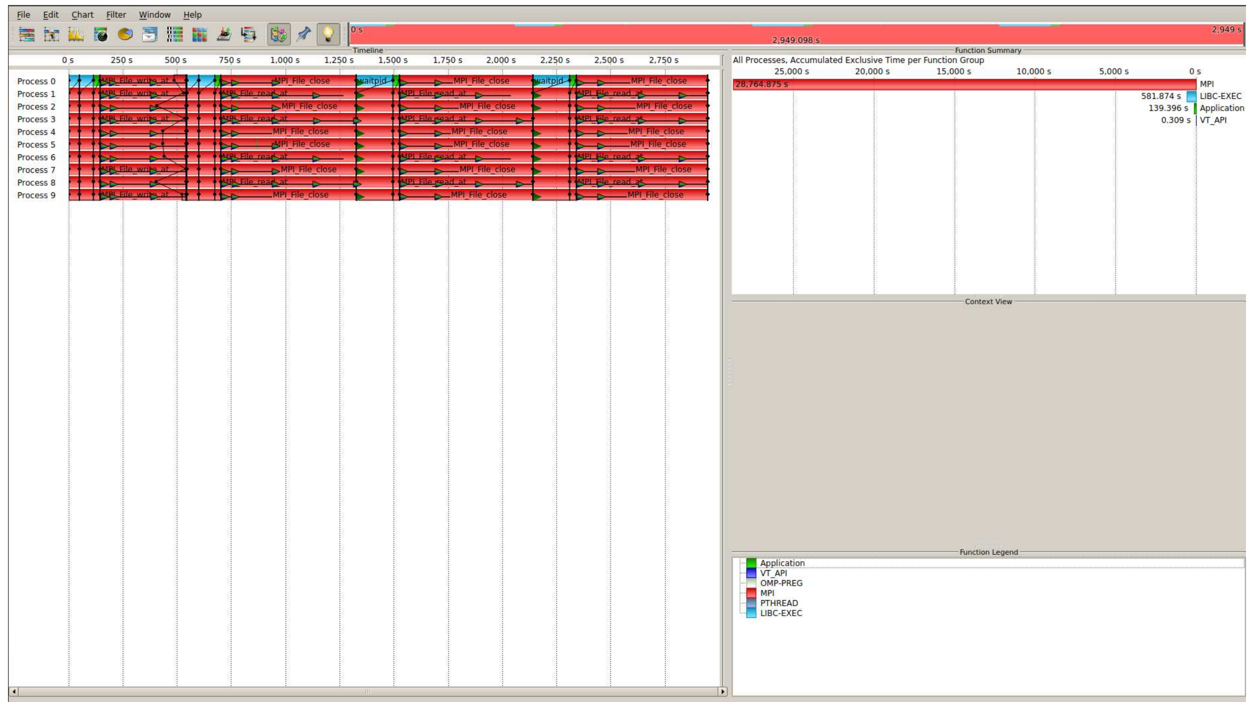

Figure 2.7.: Vampir

3. Related work

In this chapter we will discuss related work regarding the application of parallel I/O using Lustre.

As discussed in Section 2.1 and illustrated in Figure 2.1, Two-Phase I/O is an optimisation technique used by ROMIO; the goal is to perform a small amount of large contiguous accesses instead of many non-contiguous accesses (see Section 2.1). According to [DL08], this optimisation is not adequate when using Lustre as underlying parallel filesystem. When the clients are performing large contiguous accesses, each client has to communicate with many OSS, which leads to network, OSS and locking contention. We explain this issue with following example: We assume that we have a system which consists of 4 clients, 8 OSS/OSTs, a stripe size of 1 byte and stripe count of 8. The clients want to write 32 bytes in total. When using Two-Phase I/O, the file is partitioned among the clients such that one client writes the first region (bytes 1 to 8), one client the second region (bytes 8 to 16), and so on. Thus each client communicates with each OST (see Figure 3.1) and each client has to obtain 8 locks, which means that there is contention at network, OST and locking level. If the first client would write the

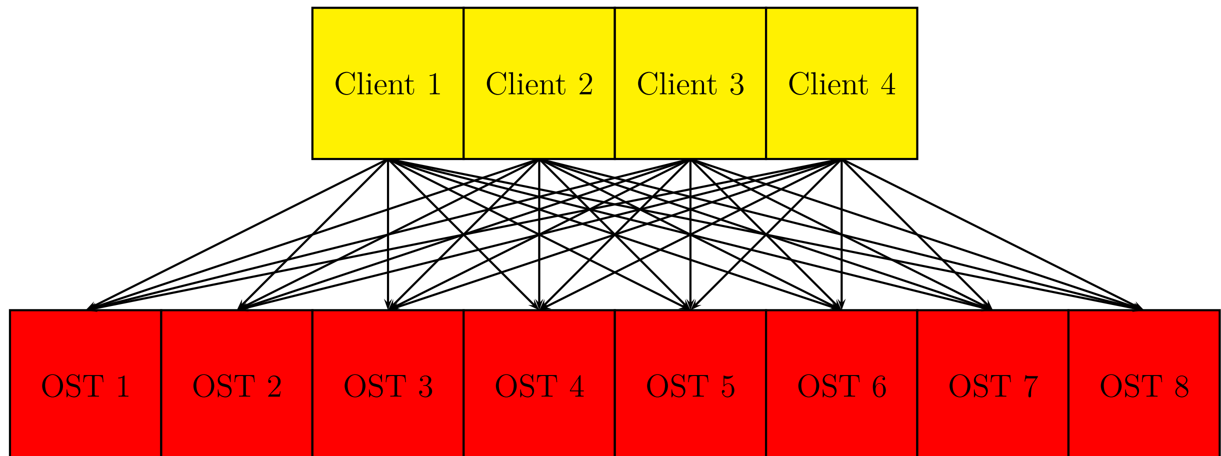

Figure 3.1.: all-to-all pattern

bytes $1, 2, 9, 10, 17, 18, 25, 26$, the second client the bytes $3, 4, 11, 12, 19, 20, 27, 28$, and so on, then the first client would only communicate with OST $1, 2$, the second with OST $3, 4$, and so on (see Figure 3.2). There would be less contention at the network and

locking level. Dickens et al. call these patterns 8-OST (or all-to-all) and 2-OST pattern, depending on the number of OSTs each client is communicating with.

In [DL08], the authors present an experimental study which has the goal to compare the performance of Two-Phase I/O with various x-OST patterns. The focus has been to find the impact of the access pattern on the performance. So, they simulated Two-Phase I/O by explicitely setting the access pattern which would result after the first phase has been applied. This ignores the cost of the redistribution of the data and sets the focus on the cost of accessing the data. The study was performed on two large HPC clusters. The results were, that the original Two-Phase I/O and other patterns which involved communicating with a lots of OSTs performed worst on both systems, while the 1-OST and 2-OST pattern provided the best performance. The authors conclude that communication with a small number of OSTs is better than with a large number.

Furthermore, the authors have implemented a user-level library called Y-Lib, which has the goal to minimise the contention for file system resources by controlling the number of OSTs with which the clients communicate([DL10]). This number depends on the particular system. The application calls this library instead of the routines from MPI-IO, and the library distributes the data using message passing such that the data access pattern confirms the respective distribution. Afterwards, the data is written to disk using POSIX write operations. Experiments have been performed on two large HPC systems, varying the number of aggregator processors used. Three different cases were distinguished on the first system.

In the first case, the data was distributed in a way among the processes such that it had to be redistributed. This means, for Y-Lib, the data has been placed in a way that each process had to write a large contiguous region(”confirming distribution”), so that Y-Lib redistributed the data. For MPI-IO, the data was set to a 1-OST pattern, so that the Two-Phase I/O redistributed it to the confirming distribution. Y-Lib outperformed MPI-IO up to a factor of 10. In the second case, the data has already been placed in the correct distribution (confirming distribution for MPI-IO and 1-OST pattern for Y-Lib), so that no redistribution was necessary. Y-Lib improved the performance up to a factor of three in this case. The last case was to force MPI-IO to use the 1-OST pattern by disabling Two-Phase I/O and data-sieving. The performance increased with a factor up to 2.

On the other system, the optimal pattern appeared to be the 2-OST pattern. In the experiment the I/O performance was compared using no redistribution, by explicitly setting the 2-OST pattern. Y-Lib was not used because using a 2-OST pattern was not already implemented. The 2-OST pattern began to outperform the conforming distribution when the count of processors exceeded a certain limit and the largest observed improvement was 36 %, significantly lower as in the first system. The authors conclude that Y-Lib can improve the performance significantly, but a benefit must not always exist, it depends on the count of processors and the particular system. They suggest to

extend Y-Lib, to find the best OST pattern using an analytic model or a set of heuristics.

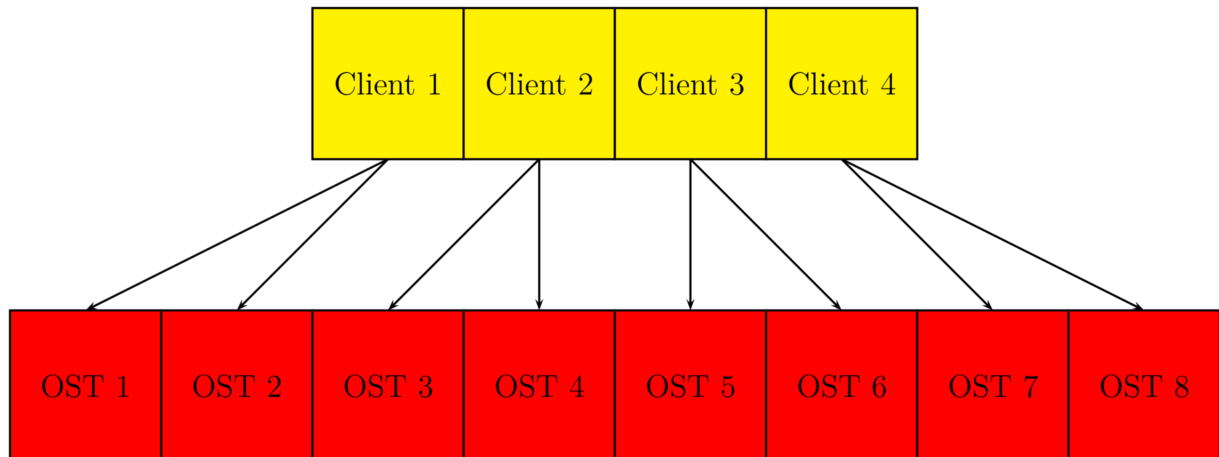

Figure 3.2.: 2-OST pattern

In [LC08], 3 modifications to the Two-Phase I/O algorithm used by ROMIO are presented and evaluated in different test scenarios using Lustre and GPFS as underlying parallel filesystem. In the original algorithm, ROMIO divides the accessed file region evenly into contiguous file domains, which are then accessed by the I/O aggregator processes. These chunks are not aligned to the Lustre stripe sizes and lock boundaries, which can impact the performance. The first modification aligns the accesses to the file system lock boundaries, which can be equal to the Lustre stripe size.

The second modification is the so called static-cycling method. It divides the entire accessed region into blocks of equal size, whose size is set to the file system's lock granularity, and assigns these blocks to the I/O aggregators in a round-robin fashion. Thus, if we have n aggregators, the blocks $i, i + n, i + 2 \cdot n, \ldots$ are assigned to the aggregator with rank i. The advantage of the approach is, that if the lock granularity size is the same as the file stripe size and there is a common divisor between the number of I/O servers and aggregators, each aggregator will communicate with the same set of I/O servers, reducing network and OSS contention. The disadvantage is, that if the number of I/O aggregators is much higher than the number of I/O servers, a lot of aggregators will communicate with the same I/O servers using interleaved accesses, which can lead to a lot of locking contention (because a lot of small lock requests have to be performed).

The third modification is called group-cycling method, and tries to eliminate these interleaved accesses. Group-cycling divides the number of I/O aggregators into groups, whose size is equal to the number of I/O servers. The aggregated access region is then divided evenly among the groups with the boundaries aligned to the stripe size. Within a group, the static-cycling method is used. In one group, each aggregator communicates with exactly one I/O server and because the file region is divided among the groups,

the access is no more interleaved. The experimental results show that none of the methods discussed above outperform the others on all file systems. Thus, the MPI-IO interface must dynamically adapt a method that works best on the target file system. Nevertheless, the results show that the group-cycling approach performs best when using Lustre in combination with collective write operations.

In [NLC08] it has been argued that a high degree of parallelism leads into high I/O contention at the servers and does not scale with the amount of processors. The authors propose a system which consists of a set of delegated I/O processes, called I/O Delegate Cache System (IODC). IODC intercepts I/O requests from the application processes, and optimises them using I/O caching and data aggregation. They evaluate their implementation using only 10 % of extra I/O processes, and discover high performance improvement in I/O bandwidth.

The authors of [YVCJ07] believe that a large stripe count can degrade the performance because of the increased protocol overhead, when clients must communicate with many OSTs, and the reduced memory cache locality, when the network buffer is multiplexed for many OSTs. They have conducted experiments which indicate this behaviour. Therefore they suggest the following approach: Instead of writing into one file, the file is split into subfiles and the processes write into the subfiles. After the file has been written, the file will be joined using the file joining feature of Lustre. According to the authors, the best I/O rate will be achieved when the file is small, each subfile does not stripe too wide, the subfiles do not overlap which would produce contention at the OSTs, and that the subfiles together cover a sufficient number of OSTs for a good aggregated bandwith. They implement these requirements in a technique called hierachical striping, which lets one determine the size of a subfile, the number of OSTs for this subfile and the start index of the OST of the subfile. Finally, they evaluate their implementation using collective management operations and concurrent read/write accesses. Their results show high performance increasement.

Finally, best practices for using HDF5 and MPI-IO with Lustre are given in [How12]; for example to set the alignment to the Lustre stripe sizes and to tune the width of the B-Tree, which is used to store the locations of the chunks. Furthermore, two software modifications to the HDF5 interface have been presented, which have been incorporated in the HDF5 release 1.8.5 . They have refactored the `H5Fflush` API routine, which has the purpose to flush all buffers associated with a file to the disk. Previously, the routine extended the file size on disk to allocate all space (even the regions which have not been written yet). This resulted in a call of the MPI-IO routine `MPI_File_set_size`, which has very poor performance on Lustre systems. Because it is not required to set the file size accurately until the file has been closed, the call has been removed from the code. The second modification is about the flushing of the metadata items. Previously, only one process wrote all metadata items to the disk, since early parallel filesystems performed poorly when multiple processes performed small I/O operations. According to the authors, this is no more true with modern file systems (as Lustre),

which have multiple storage servers and can handle multiple small requests at the same time. The interface has been modified to divide up the metadata into groups and each process flushes one group to the disk. These modifications and best practices have been evaluated in combination with the modifications from [LC08], which are incorporated in the Cray Message Passing Toolkit 3.1 . The results show that the performance is much better than using unoptimised configurations.

4. Evaluation

This chapter is dedicated to an extensive analysis of the NetCDF-4 and HDF5 interfaces concerning parallel I/O access. We compare many I/O configurations and analyse the full I/O path (POSIX, MPI-IO, HDF5 and NetCDF-4). We distinguish the access of the data into two modes: disjoint and interleaved. Disjoint means that each client accesses a large contiguous region of the data, as illustrated in Figure 4.1.

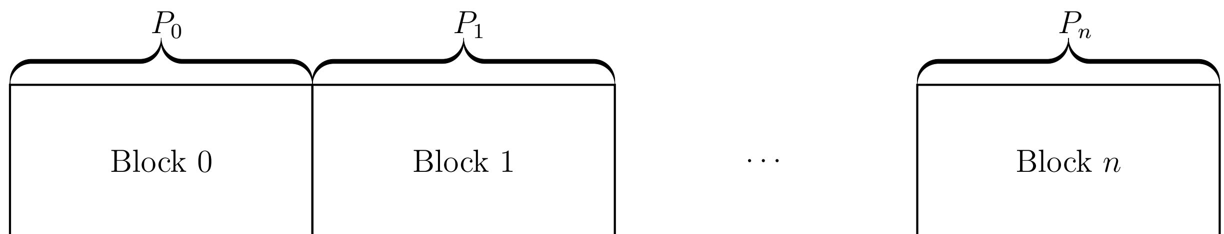

Figure 4.1.: Disjoint pattern

We speak of interleaved access when each client accesses a large non-contiguous region. This pattern is illustrated in Figure 4.2. One has to consider, that we refer to logical and not physical access: The accesses described above are related to the calls in the application level, which does not mean that the physical access is performed in the same way. For example, when using chunking the data is physically spread in a non-contiguous way even when using logically contiguous accesses.

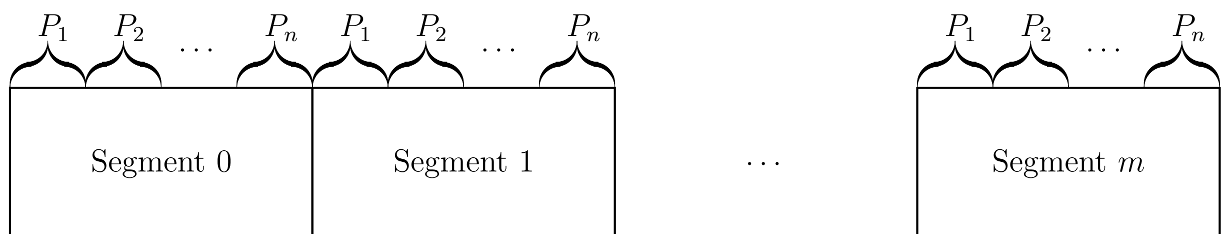

Figure 4.2.: Interleaved pattern

We analyse the performance using contiguous and chunked data layout, with a one-dimensional data shape. Furthermore, we are interested in the impact on the perfor-

mance when a multi-dimensional data shape is used. For this purpose, we also analyse the performance with a two-dimensional data shape.

4.1. Testbed specification

We use the software libraries and versions stated in Table 4.1.

Software	Version
HDF5	1.8.11
Open-MPI	1.6.4
Lustre	2.5
NetCDF-4	4.1.3
IOR	3.0.1

Table 4.1.: Software versions

Our system consists of 10 OSS nodes that host the OSTs. For simplicity's sake we will only speak of OSTs instead of OSS in this thesis. Furthermore, one of the servers mentioned does also host one MDS. Each of these server nodes has Intel® Xeon™ Sandy Bridge E-1275 CPUs (3.4 GHz, 4 cores total), 16 GB DDR3/PC1333 ECC RAM, 3 × 2 TB SATA2 Western Digital WD20EARS HDDs, a 160 GB SATA2 Intel 320 SSD and 2 × Intel® 82579LM/82574L Gigabit Ethernet NICs. They use CentOS 6.5 with Linux 2.6.32-358.18.1.el6_lustre.x86_- 64. We use 10 client nodes. These nodes each have 2 × Intel® Xeon™ Westmere EP HC X5650 CPUs (2.66 GHz, 12 cores total), 12 GB DDR3/PC1333 ECC RAM, a 250 GB SATA2 Seagate Barracuda 7200.12 HDD and 2 × Intel® 82574L Gigabit Ethernet NICs. They use Ubuntu 12.04.3 LTS with Linux 3.8.0-33-generic.

All nodes are interconnected via Gigabit-Ethernet. We can therefore calculate the theoretical maximum performance: The throughput is bounded by the network and we have 10 client nodes each bounded by 1000 MBit/s which equals 125 MB/s[1]. TCP/IP is used as underlying network transport protocol stack and adds overhead to the pure data transport. In [Kun13], researchers of the research group Scientific Computing at the University of Hamburg, which use the same network environment, measured an available bandwith of 117 MB/s due to protocol overhead,etc. Thus the maximum is

$$10 \times 117\text{MB/s} = 1170\text{MB/s} \approx 1125\text{MiB/s}$$ [2]

[1]MB/s stands for Megabyte per second, thus 10^6 bytes per second

[2]MiB/s stands for Mibibyte per second, which equals 2^{20} bytes per second

4.2. Modifications to IOR

We have modified IOR to fit our analysis purposes. IOR is written in C and has one main source file, which implements several functions for creating tests, printing the results, doing simple statistics and running tests. Tests can write/read data and check that writing/reading was correct. IOR has one driver for each API, which is realised in a source file that implements the so called abstract ior interface (aiori) . This interface (implemented as a C-struct, see Listing 4.1) declares functions for creating, closing and opening a file, transferring data, etc. The main file binds at startup to the concrete implementation and calls these functions when performing the test. This abstract design lets one easily add drivers for other APIs without changing the main file.

```
1  typedef struct ior_aiori {
2          char *name;
3          void *(*create)(char *, IOR_param_t *);
4          void *(*open)(char *, IOR_param_t *);
5          IOR_offset_t (*xfer)(int, void *, IOR_size_t *,
6                                IOR_offset_t, IOR_param_t *);
7          void (*close)(void *, IOR_param_t *);
8          void (*delete)(char *, IOR_param_t *);
9          void (*set_version)(IOR_param_t *);
10         void (*fsync)(void *, IOR_param_t *);
11         IOR_offset_t (*get_file_size)(IOR_param_t *,
              ↪ MPI_Comm, char *);
12 } ior_aiori_t;
```

Listing 4.1: aiori struct

NetCDF support is integrated by a driver for the parallel interface pnetcdf. In the words of [LLC+03], “pnetcdf is a parallel interface for writing and reading netCDF datasets. This interface is derived with minimum changes from the serial netCDF interface but defines semantics for parallel access and is tailored for high performance.” This interface does only support NetCDF files of a version which is less than 4. Therefore, we have added a driver for the parallel version of NetCDF-4.This driver provides the nofill option, chunking, the usage of unlimited dimensions and multiple dimensions. Hence we have added the options *chunkSize*, *unlimited*, *dimx*, *dimy*, *chunkx* and *chunky*. The *chunkSize* option specifies the size of the chunks when using only one dimension. If *unlimited* is set to 1, the first dimension will have size unlimited, which means that the variable, which holds the data that is written, can be arbitrarily extended. *dimx* and *dimy* are required when two-dimensional data layout is desired. *dimx* specifies the size of the first and *dimy* that of the second dimension. The variable is defined using the datatype `NC_BYTE`, which means that the product of *dimx* and *dimy* equals the aggregated filesize[3]. *chunkx* and

[3] This is not exactly true, because the metadata adds overhead. But this overhead is so small, that it is negligible.

chunky specify the size of the chunks respectively. If none of the parameters *unlimited*, *dimx*, *dimy*, *chunkx* and *chunky* are used, the size of the only dimension equals the aggregated filesize.

Listing 4.2 shows our implementation for transferring data. Lines 14 to 18 show the respective calls for putting/retrieving data. `fd` is the identificator of the NetCDF file we are using and `var_id` the identificator of the variable. `start` and `count` are one-dimensional arrays, whose size corresponds to the dimension of the variable. The `buffer` contains the data. Lines 3 to 6 are related to the two-dimensional case. We always write the full transfer size along the second dimension, thus the count of the first dimension is set to 1 and the count of the second dimension to the transfer size. Furthermore, it is required that the *dimy* parameter is a multiple of the transfer size and the number of tasks which are participating. The start offset is calculated by taking into consideration that the second dimension is the fastest-varying dimension. Lines 10 and 11 show the one-dimensional case. `start` is set to the offset and `count` to the transfer size.

```
1  if( param->dimx && param->dimy )
2  {
3      start[0] = (size_t) (param->offset / param->dimy);
4      start[1] = (size_t) (param->offset % param->dimy);
5      count[0] = (size_t) 1;
6      count[1] = (size_t) (param->transferSize);
7  }
8  else
9  {
10     start[0] = (size_t) param->offset;
11     count[0] = (size_t) (param->transferSize);
12 }
13 /* access the file */
14 if (access == WRITE) {  /* WRITE */
15     NC4_CHECK(nc_put_vara(*(int *)fd, var_id, start, count,
         ↪ (char*) buffer), "cannot write into variable");
16 } else {                /* READ or CHECK */
17     NC4_CHECK(nc_get_vara(*(int *)fd, var_id, start, count,
         ↪ (char*) buffer), "cannot read from variable");
18 }
```

Listing 4.2: NC4 data transfer algorithm

We have modified the HDF5 driver to support multiple segments in one dataset. The original version created a new dataset for each segment. We were required to modify this, because our analysis purposes require interleaved access in one dataset (see Section 4.5). Analogous to the NetCDF-4 driver, we have implemented chunking, unlimited dimension and two-dimensional data layout. Finally we have integrated calls to restart the Lustre system.

4.3. Experimental design

Except the stripe count parameter (see Section 2.4), we use Lustre in default configuration. This means that server-side locking is disabled and the stripe size parameter is set to 1 MiB. We set stripe count to -1, which means that a file is spread over all OSTs.

Table 4.2 shows the IOR parameters which we are using in all of our experiments. In each I/O configuration we write and read back data, repeating 3 times. IOR calculates the write/read bandwidth in the following manner ([iora]): "IOR performs get a time stamp START, then has all participating tasks open a shared or independent file, transfer data, close the file(s), and then get a STOP time. The calculated bandwidth is the amount of data transferred divided by the elapsed STOP-minus-START time." The figures of the experimental results are given in Appendix A. Our plots do always show the mean of the different measurements, if not stated otherwise. We have run several tests with different choices for the *memoryPerNode* parameter and it seems that this parameter does not have a lot of impact on the performance; thus we set the parameter to 0 (which means no additional memory allocation). To ensure the independence of each run, we restart the Lustre system before each write or read test and wait 30 seconds before performing the test. Restarting includes cleaning the caches of the client and server nodes. Prefilling, as discussed in Section 2.2, slows down the performance approximately by the half; because we know this in advance it makes no sense to use this option in our experiments. Thus we use the option *noFill*.

We use 1 process per client node, because we believe that this is satisfying to saturate the network interface, which means that adding more processes would only result in contention and worse performance. Furthermore, we set the parameter *reorderTasksConstant* to 1, to prevent that the data is read back from the cache (see Section 2.5). The goal of our experiments is to discover the full potential of the system. Therefore we use the full amount of client nodes and set *numTasks* to 10. We write/read 20 GiB[4] per node. This exceeds the available memory on the client nodes (see Section 4.1) and prevents that the data is only written to memory and not to the disk storage. As we use 10 nodes, the aggregated filesize is equal to 200 GiB. If not stated otherwise, we set *transferSize* to 1 MiB and believe that this is a good value, because according to [OI14] the Lustre clients internally send all their data in 1 MiB chunks, if it is possible. Furthermore the default Lustre stripe size equals 1 MiB. When using chunked I/O, we set the chunk size to 1 MiB, too. If not stated otherwise, we align the beginning of the data section to the Lustre stripe sizes. This also includes the physical start adress of the chunks, if chunked I/O is used. We therefore set the parameter *setAlignment* to 1 MiB. It has to be mentioned, that this mechanism does not work when using the NetCDF-4 API, because the aligning is not implemented. We have implemented this feature in this thesis. For more details see Section 5.1.

[4]GiB stands for Gibibyte, which equals 2^{30} bytes

If not mentioned differently, we run every test case once with independent I/O and once with collective I/O. For collective I/O, we disable the Two-Phase I/O optimisation (see Figure 2.1), because it makes no sense for our analysis purposes. This optimisation tries to turn non-contiguous accesses into contiguous ones by reordering the data accesses. This is useless when analysing the disjoint pattern, because the clients are already accessing large contiguous regions. Furthermore, if we would use this optimisation with the interleaved pattern, the optimisation would turn the interleaved access into a disjoint access, which would not be the scenario which we want to analyse. The MPI-IO driver of IOR uses the routines `MPI_File_write_at_all` and `MPI_File_read_at_all` in order to write/read data collectively. According to [ROM] these routines perform independent I/O, when Two-Phase I/O is disabled. For this reason we do not use MPI-IO in combination with collective I/O in our tests. The HDF5 API internally uses fileviews when performing collective I/O; this results in a call of `MPI_File_set_view` which is collective. Thus synchronisation will be performed which affects the I/O behaviour. We are therefore interested in the comparision between independent and collective I/O when using NetCDF-4 and HDF5.

readFile	=	1
writeFile	=	1
repetitions	=	3
reorderTasksConstant	=	1
memoryPerNode	=	0
interTestDelay	=	30
noFill	=	1
numTasks	=	10
transferSize	=	1m
setAlignment	=	1m

Table 4.2.: Basic IOR configuration

4.4. Disjoint pattern

As we write 20 GiB per process, in all tests of this section, additionally to the Basic IOR configuration stated in Table 4.2, we use the IOR configuration stated in Table 4.3.

segmentCount	=	1
blockSize	=	20g

Table 4.3.: IOR configuration - disjoint pattern

4.4.1. Contiguous data layout

Figure 4.3 shows the results of a test using contiguous data layout. We have plotted the minimum and maximum throughput, because many results have a high variation when using independent I/O. We believe that the reason lies in the combination of the competition on the OST and network resources with the lack of synchronisation when using independent I/O.

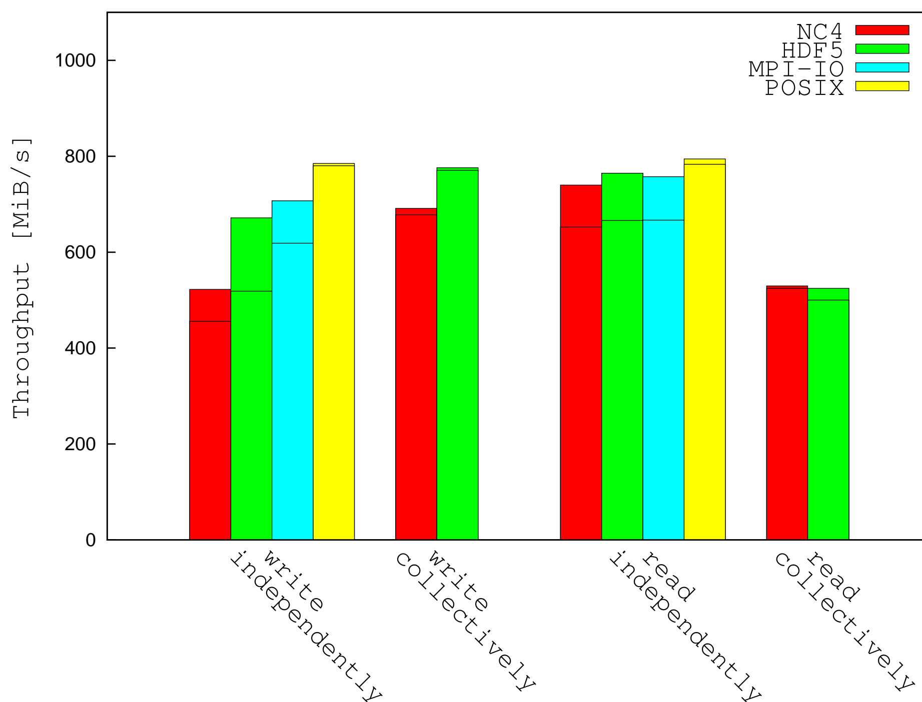

Figure 4.3.: Disjoint pattern

We see that the results are far away from the theoretical maximum stated in Section 4.1. We are sure that this can be explained by the contention on the OSTs and network resources, because the access pattern is an all-to-all OST pattern; each client node is communicating with each OST node. When analysing the results using independent I/O, we observe that reading performs better than writing[5], especially in the case of MPI-IO, HDF5 and NetCDF-4. We have the following explanation for this issue: Figure 4.4

[5]If not stated differently, we mean for each API and each value (minimum and maximum).

shows a trace, using the tool Vampir (see Section 2.6), of an example run with POSIX, which indicates that some clients are deferring their respective writes, which leads to the inconsistent figures and low data rates. We have analysed the log files on the client nodes and they indicate that some OSTs are not able to handle all the requests to write data from the clients. Therefore some clients are deferring their respective writes. We believe that the reason lies in the combination of the access pattern (all-to-all) and the lack of synchronisation when using independent I/O; the write requests of some clients arrive earlier and in a larger flow at the OSTs, which then prefer those clients. The other clients loose the connection to the sever, due to a timeout, because their write requests have not been responded to.

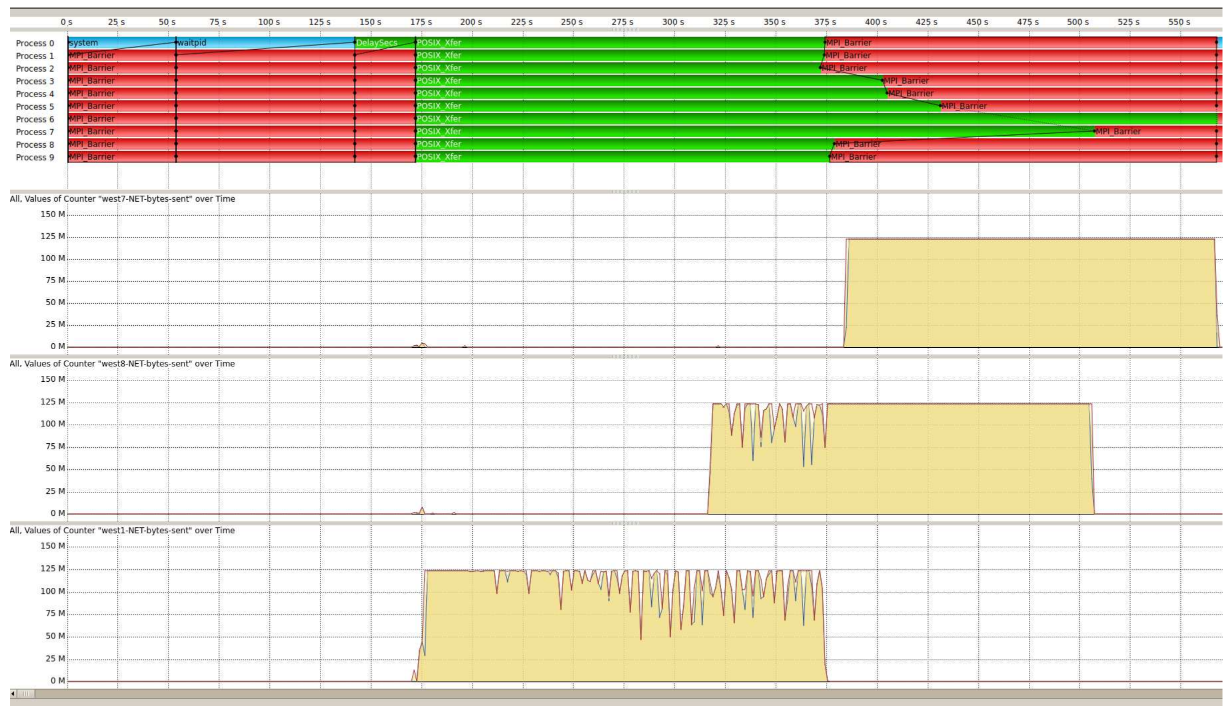

Figure 4.4.: Some clients defer writes

POSIX performs better than MPI-IO, which performs better than HDF5 which itself performs better than NetCDF-4. The difference is especially strong when comparing the peak performance of HDF5 with NetCDF-4, the minimum value of MPI-IO with HDF5 and MPI-IO with POSIX; but we cannot conclude if this is random behaviour (due to the inconsistency) or related to the APIs. When reading, POSIX achieves higher figures than the other APIs, but the difference is very small if we consider the peak performance. The difference between minimum performance is strong, but again, this could be a coincidence. The other APIs perform almost equally, which means that the API overhead is not strong in this case.

In the case of collective I/O, NetCDF-4 and HDF5 attain higher data rates when writing.

HDF5 achieves better results when writing than NetCDF-4, which could be related to the API overhead or the lack of alignment in the NetCDF-4 API. Furthermore, we note that HDF5 achieves similar data rates as POSIX in independent mode, which is an indicator that the we have measured the maximum which is possible for HDF5. When reading collectively, both APIs perform similarly.

Finally, if we compare collective I/O against independent I/O, we observe that writing performs better using collective I/O than when using independent I/O, surely due to the synchronisation which prevents the deferring of the write requests of the clients, as explained above. Reading with NetCDF-4 and HDF5 achieves better results when independent I/O is applied.

The goal of the following test is to discover the impact of the transfer size on the performance. Because of the inconsistent behaviour when using independent I/O in combination with this access pattern, we constrain the test to collective I/O. Figure 4.5 shows the results. Writing performs better than reading until a large transfer size is reached (128 MiB), then reading performs better than writing. Furthermore, read performance increases monotonically with the transfer size for both APIs. We believe that the reason lies in the fact that write requests can be aggregated and cached by the OSTs, so they can be written out in a performant way. This is not possible when reading, because the OSTs have to deliver the data immediately which can implicate a lot of seeking, especially when the transfer size is small. This behaviour hampers the throughput. When the transfer size is large, the amount of seeks which have to be performed is relatively small; thus reading benefits from a larger transfer size.

Writing with HDF5 performs relatively constant. Only with small transfer sizes (128 and 256 KiB[6]) the performance is very poor. We believe that such small transfer sizes imply a larger communication overhead, because more network packets have to be transmitted (which itself contain metadata). Furthermore, HDF5 outperforms NetCDF-4 until a transfer amount of 4 MiB is reached. The reason lies in the lack of alignment in the NetCDF-4 API. Unaligned writing implicates that some write requests are split up to multiple OSTs, which results in smaller chunks which have to be processed on the OSTs for the respective requests. This overhead is especially significant when the transfer size is small, because the fraction of requests which have to be split up is larger. Reading with either HDF5 or NetCDF-4 performs almost equally. Unaligned I/O does not seem to degrade the performance, even when using small transfer sizes. The reason could be the fact, that reading with small transfer sizes does already perform poorly, and the splitting of the requests has no more influence on the poor performance.

Finally, we analyse the impact of the API overhead of NetCDF-4 when using different transfer sizes. Therefore we compare NetCDF-4 with HDF5 using unaligned I/O. The results are illustrated in Figure 4.6. When writing, both APIs perform almost equally. When reading with larger transfer sizes (> 4 MiB), HDF5 performs slightly better, but

[6]KiB stands for Kibibyte, which equals 2^{10} bytes

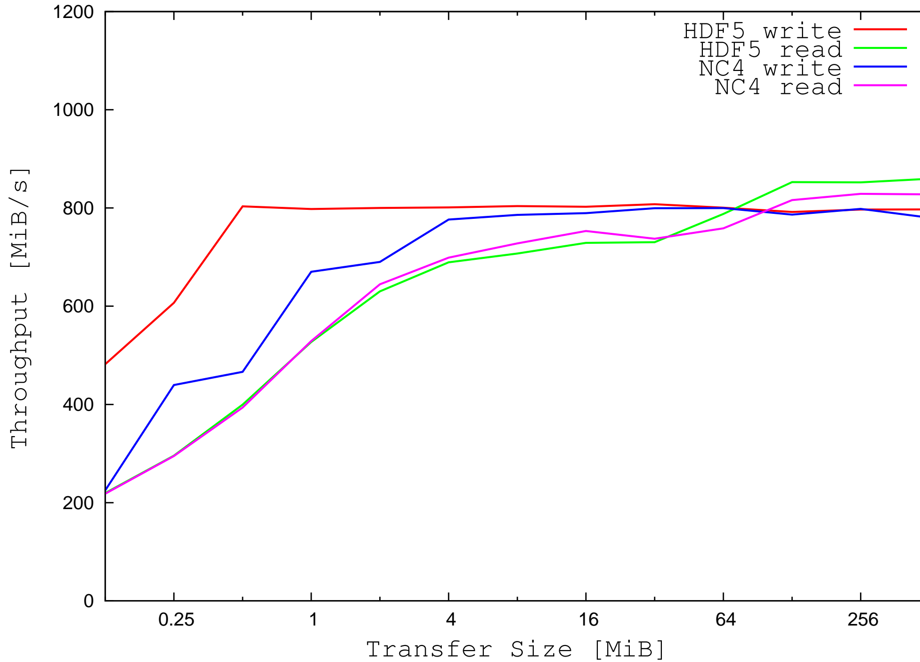

Figure 4.5.: Varying transfer size

the API overhead of NetCDF-4 is not significant.

4.4.2. Chunked data layout

In the following test we analyse the performance using chunked data layout with various chunk sizes. We set the transfer size to 512 MiB, because our previous tests have indicated that the performance benefits from a large transfer size. Furthermore, we use collective I/O to attain figures with low variation. Figure 4.7 shows the results. Overall, whether writing or reading, the performance increases with the size of the chunks and stays relatively constant at a particular chunk size. Small chunk sizes imply a larger amount of chunks which implies a larger amount of metadata that has to be handled. As mentioned in Section 2.2, the locations of the chunks are stored in a B-tree. Halving the chunk sizes increases the size of the B-tree by double. This overhead slows down the performance dramatically, especially when writing, because the B-tree has to be calculated and written to the header. NetCDF-4 performs worse than HDF5 with small chunk sizes. The reason is not the lack of alignment in the NetCDF-4 API, because we have performed tests with HDF5 unaligned that achieved the same figures as HDF5 with aligned I/O. Therefore, the API adds overhead in this case. When the chunk size is sufficiently large, the results are analogous to the contiguous case using 512 MiB as transfer size (see Figure 4.5). The reason is that the metadata overhead, mentioned above, is no longer important and the same access pattern is used: Large contiguous

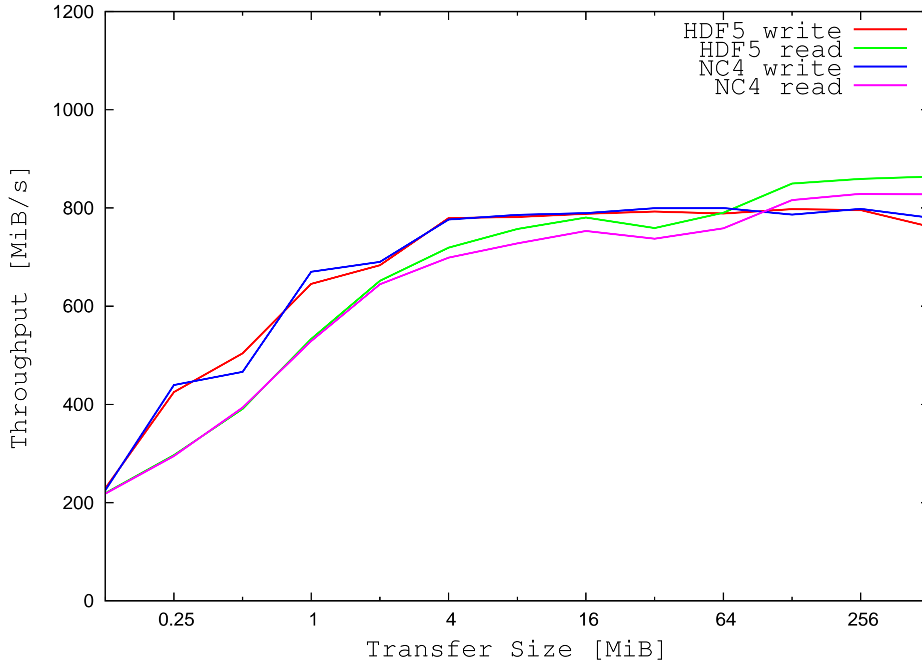

Figure 4.6.: Comparing NetCDF-4 with HDF5 unaligned

regions are accessed by each process.

4.4.3. Two-dimensional data layout

Finally, we analyse the disjoint scenario using two-dimensional data layout. We use the IOR configuration stated in Table 4.4; thus, the first dimension has a size of 2 KiB and the second 100 MiB. The results are shown in Figure 4.8. As in the one-dimensional case, the results are far away from the theoretical maximum. More precisely, the results are nearly the same when the figures are consistent, which is the case when collective I/O is used. In the case of inconsistency the peak performance is always in similar regions. The reason is, that the underlying access pattern is the same: a large contiguous region will be accessed, leading into an all-to-all pattern.

dimx	=	2k
dimy	=	100m

Table 4.4.: IOR configuration - Two dimensions

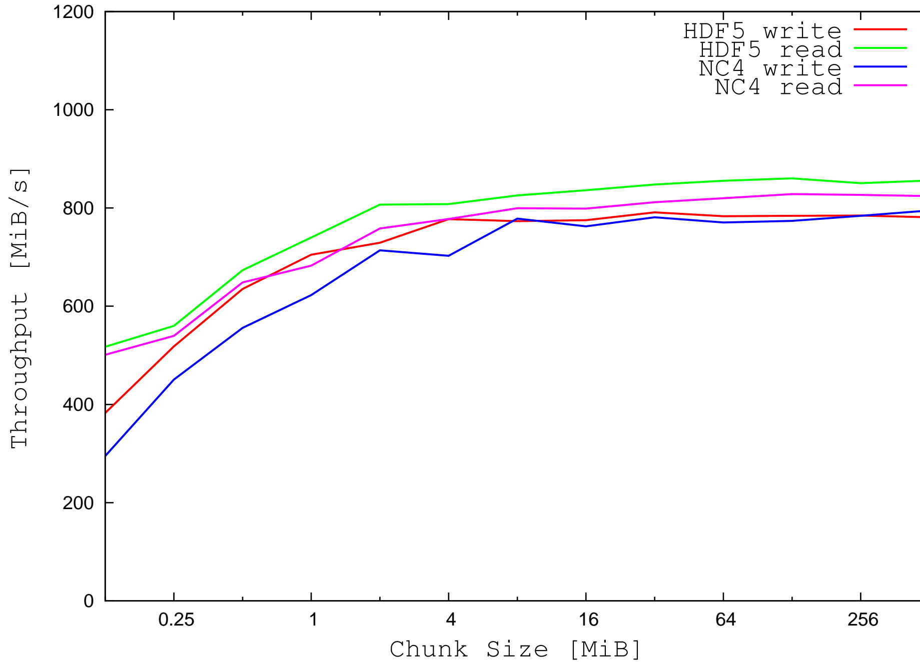

Figure 4.7.: Chunked layout

4.4.4. Summary

We conclude that this access pattern is not suited if maximal throughput performance is desired: All our tests achieve data rates which are far away from the maximal theoretical performance mentioned in Section 4.1. The reason is the competition on network and OST resources, because all clients are communicating with all server nodes. We explained this suspicion in Chapter 3.

Many results have a high variation when using independent I/O. We believe that the reason lies in the lack of synchronisation combined with the access pattern. Furthermore, write performance with independent I/O is very poor in some cases, because some clients are deferring their respective writes, because the server is not capable to respond to the respective requests.

The highest throughput (whether reading or writing) is achieved when using large transfer sizes. This is especially important when reading, because the overhead induced by the seeking is very little in this case. Reading does even perform better than writing with very large transfer sizes. Furthermore, for small transfer amounts, HDF5 outperforms NetCDF-4 when writing. The reason lies in the unaligned data access of NetCDF-4. The NetCDF-4 API adds almost no overhead with this access pattern, as HDF5 with unaligned I/O performs almost identical to NetCDF-4 when writing and slightly better when reading with large transfer amounts.

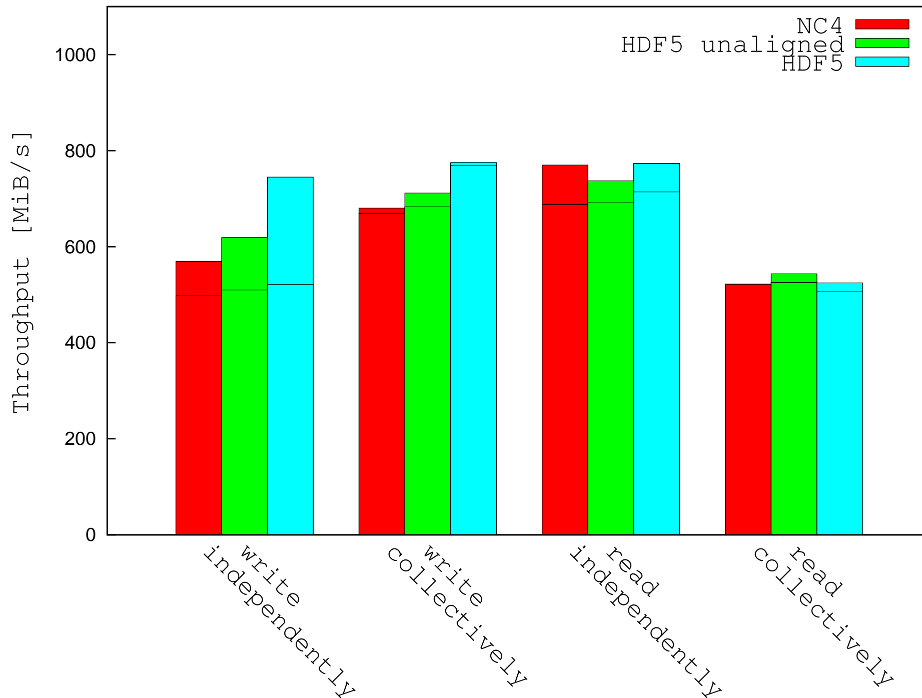

Figure 4.8.: Disjoint pattern multiple dimensions

Chunked I/O benefits from large chunk sizes, as the metadata overhead using small chunks slows down the performance dramatically. When using large chunk sizes in combination with a large transfer size (512 MiB), the results are equal to the contiguous case. Finally, the dimensionality of the data layout appears to have no real impact on the performance behaviour.

4.5. Interleaved pattern

In this section we will analyse different x-OST patterns. The term has been discussed in Chapter 3 and it has been taken from [DL08]. In our scenarios an x-OST access means, that each client node is communicating with exactly x OST nodes in the whole test. We define a pattern to also be x-OST when using chunked layout, if it is x-OST when using contiguous layout; though chunked I/O spreads the data into independent locations, which means that we cannot determine the number of OSTs the client is communicating with. Furthermore a pattern is also x-OST when using NetCDF-4, although the access is not aligned, which means that it is communicated with more than x OSTs.

When using x-OST access, there exist groups with competing clients. E.g. in the 2-OST pattern with 10 client nodes evenly spread over the OSTs, each OST will communicate with 2 clients. These 2 clients compete for the resources. In this section we want to

analyse the impact of this competition on the performance.

4.5.1. Contiguous data layout

We analyse the 1-OST pattern using the IOR configuration stated in Table 4.5 and contiguous data layout. Figure 4.9 shows the results. Writing or reading with POSIX and MPI-IO performs nearly identical, achieving almost the theoretical maximum. The 1-to-1 communication between clients and servers prevents competition on OST and network resources. Contrary to POSIX and MPI-IO, HDF5 performs better when reading than when writing; reading independently performs even equal to POSIX and MPI-IO, but writing performs significantly worse. We believe that the write performance is affected by the API overhead induced by the HDF5 API, whereas reading benefits from the ability of the clients to read-ahead data. Furthermore, HDF5 benefits from independent I/O. Collective I/O adds unnecessary synchronisation between the clients when performing a 1-to-1 communication, which slows down the performance. Contrary to the little overhead of the layering induced by HDF5, NetCDF-4 performs much worse. The reason is that the alignment of the data is not supported in NetCDF-4, leading to unaligned data access, which results in communication with more than one OST. Reading (either collective or independent) and writing independently achieves about half of the performance of HDF5, which could be explained by the fact that due to the unaligned I/O with each request two OSTs are involved. Writing in collective with NetCDF-4 slows the performance down dramatically.

segmentCount	=	20k
blockSize	=	1m

Table 4.5.: IOR configuration - 1-OST pattern

The goal of the following test is to analyse the impact of a large transfer size (512 MiB) when using the 1-OST pattern. The IOR benchmark requires that the transfer size is not larger than the block size, which means that we cannot use IOR for this particular scenario. We have written benchmarks for HDF5 and MPI-IO, which use this access pattern and call the respective write/read routines with 512 MiB as transfer size. The POSIX interfaces do not provide the ability to write/read using non-contiguous accesses, which means that for all APIs the respective calls are split into contiguous calls when the data arrives at the POSIX layer. Furthermore, the NetCDF-4 API provides the ability to use non-contiguous calls; but we have looked into the implementation of the corresponding routine, which shows that afterwards all calls are directly converted into contiguous calls of another high-level NetCDF-4 routine. This means that it makes no sense to write a benchmark using this routine, because it directly results in contiguous calls of the NetCDF-4 API. Therefore, we have only written benchmarks for HDF5 and

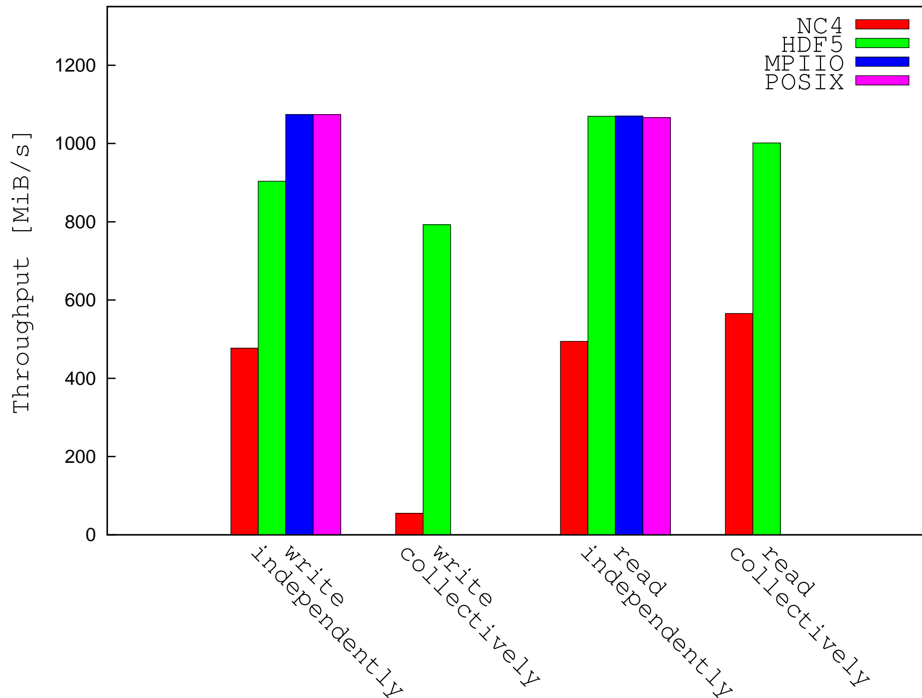

Figure 4.9.: 1-OST pattern

MPI-IO, which have the purpose to test the decrease of the API overhead when using large transfer sizes. We have to note that our implementation of the MPI-IO benchmark uses the routine `MPI_File_set_view`, which is collective across all processes. This means that each call is synchronised across all processes, contrary to the implementation of the MPI-IO driver of IOR when using independent I/O.

Figure 4.10 shows the results in comparison with the results using 1 MiB as transfer size. We observe that the performance of MPI-IO does not change, which indicates that the maximal reachable performance with MPI-IO has already been achieved. Except when reading independently, the performance increases for HDF5 when using 512 MiB as transfer size. Reading collectively performs equal to MPI-IO, in this case. Furthermore, writing independently and collectively performs equal, but still worse than MPI-IO. The performance enhancement is especially strong for writing using collective I/O. The reason for the performance enhancement in the case of collective I/O is the smaller synchronisation overhead, as a larger transfer size results in fewer calls of the write/read routine, which itself results in fewer synchronisation calls. We conclude that the API overhead does still exist for writing with HDF5, but using large transfer sizes decreases this overhead.

Figure 4.11 shows the results when using the 2-OST pattern. The IOR configuration is stated in Table 4.6. We have plotted the minimum and maximum as some figures show high variation when using independent I/O. In the case of writing with independent I/O,

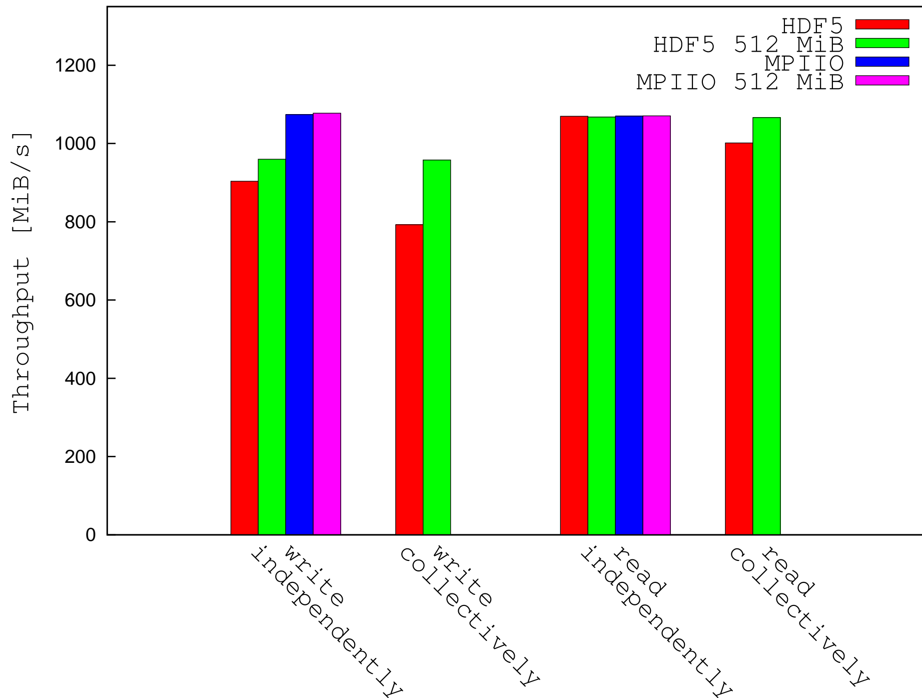

Figure 4.10.: 1-OST pattern, comparing transfer size

all APIs achieve figures that lie in the region of about the half of the data rate of the 1-OST pattern. POSIX, MPI-IO and HDF5 perform similarly; NetCDF-4 is performing worse, due to the lack of aligning. When writing collectively, the performance slows down dramatically, and NetCDF-4 and HDF5 perform similarly. We believe that the reason for these poor figures is the competition on the OST and network resources, as each client competes with another client for two OSTs, combined with the interleaved access on each OST. Each OST writes one stripe of one client after the stripe of the second client, and so on. We speculate that this access pattern is much more expensive when than using the disjoint pattern, where each OST can allocate a large contiguous region on the underlying storage system, and has few overhead regarding the consistency of the data (locking, etc). The synchronisation of collective I/O leads to similar arrival times of the respective write requests. We suspect that this is a great disadvantage which leads to the poor figures. In the case of independent I/O, several write requests of one client can arrive independently, which allow greater possibility to optimise the access to the underlying storage system, as more data layout information is available.

We observe a high variation of the results when reading independently; furthermore the minimum performance is significantly lower than when using collective I/O. We have the following explanation for this issue: The seeking overhead on the OSTs is very high using interleaved access, if the read requests arrive in an unfavourable way. Figure 4.12 illustrates an access of two clients on an OST. Consider, that all read requests of Client A arrive before the requests of Client B. If each client accesses n stripes, the OST has

to seek $2 * n$ times. If the requests would arrive at the same time, the OST could mix the requests and would only be required to seek maximal n times. Therefore, reading benefits from collective I/O in this case, as the read requests are synchronised and the seeking overhead decreases. The minimum read performance of HDF5, MPI-IO and POSIX is similar; achieving less than half of the performance of the 1-OST pattern. NetCDF-4 performs better; the unaligned access leads to more clients accessing one OST, which allows the mixing of more requests and decreases the seek overhead. One run of HDF5 is an outlier, performing equally to the collective I/O performance of HDF5. Reading collectively with NetCDF-4 performs slightly better than independently, due to the benefit gained from synchronisation.

segmentCount	=	10k
transferSize	=	2m
blockSize	=	2m

Table 4.6.: IOR configuration - 2-OST pattern

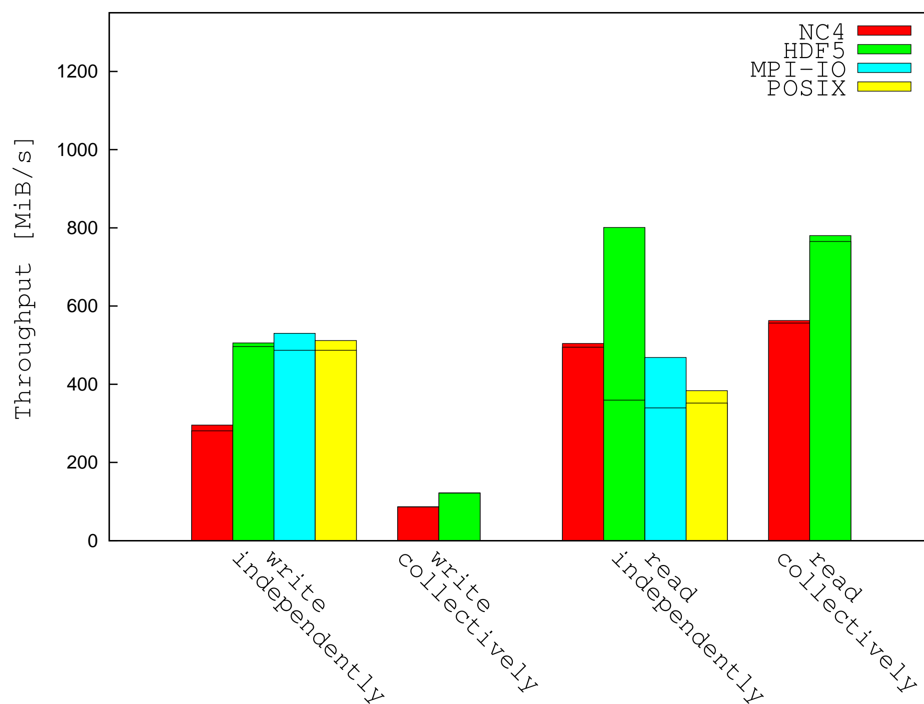

Figure 4.11.: 2-OST pattern

In the following test we explore the impact of a large transfer size (512 MiB) using the 2-OST pattern. We use our self-developed benchmarks, which have been mentioned above. Figure 4.13 shows the results. Writing with HDF5 using collective I/O benefits strongly from the large transfer size, as the synchronisation overhead decreases. Writing

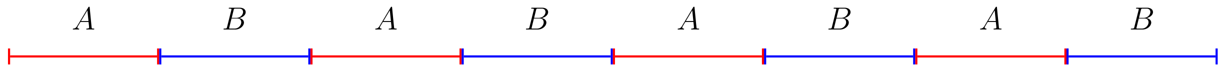

Figure 4.12.: Interleaved access

independently performs almost equally. The maximum read performance does even perform worse, which is significantly expressed when using independent I/O. But due to the inconsistency when reading, we cannot conclude if this is hazard or not. As mentioned above, our MPI-IO benchmark internally uses a collective MPI-IO routine each time before the data is written/read. This is the reason why the results of MPI-IO are very similar to HDF5 using collective I/O. The write performance decreases and the read performance increases with our benchmark. But we believe that the latter increasement is due to the synchronisation and not due to the larger transfer size. Finally, we conclude that this access pattern does not benefit from large transfer sizes, except when writing collectively.

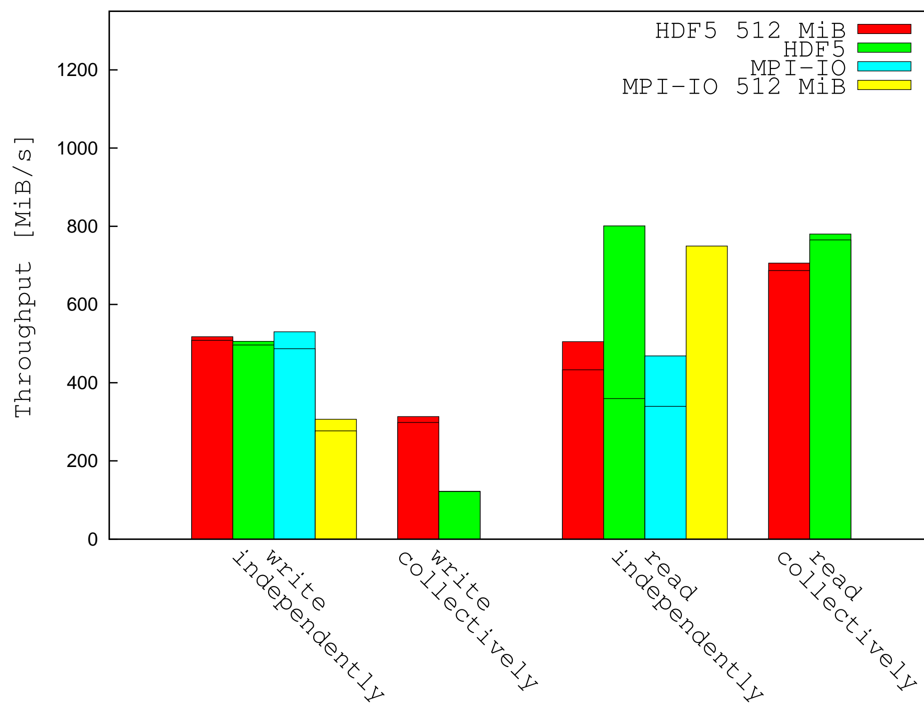

Figure 4.13.: 2-OST pattern, comparing transfer size

Finally, we analyse the 5-OST and 10-OST pattern in comparison with the 1-OST and 2-OST pattern. The IOR configurations are shown in Table 4.7 and 4.8. We omit the tests for POSIX, MPI-IO and HDF5, as we do not believe to attain performance increasement over the 1-OST pattern. The results are illustrated in Figure 4.14. When writing independently, the 1-OST pattern outperforms the other patterns, because, as explained

above, the interleaved access performs very poorly in combination with competition on the OSTs. Writing collectively performs very poor, but the performance increases slightly with the number of OSTs it is communicated with. We believe that the reason lies in the increasement of the transfer size which we use in our tests, which leads into fewer synchronisation calls. Reading independently benefits from the communication with more OSTs. We believe that the reason is that more clients per OST gives each OST the possibility to optimise the access and to decrease the seek overhead. Reading collectively performs moreover constant, except the 5-OST pattern, which performs worse.

segmentCount	=	4k
transferSize	=	5m
blockSize	=	5m

Table 4.7.: IOR configuration - 5-OST pattern

segmentCount	=	2k
transferSize	=	10m
blockSize	=	10m

Table 4.8.: IOR configuration - 10-OST pattern

4.5.2. Chunked data layout

In the following tests we analyse the performance behaviour of HDF5 and NetCDF-4 with chunked data layout, varying the number of OSTs it is communicated with. We use the IOR configurations which are stated in the Tables 4.5, 4.6,4.7 and 4.8. Furthermore, we set the chunk size to the maximum meaningful value, which is the value of the block size, because larger chunk sizes decrease the meta data overhead, as discussed in Subsection 4.4.2. Figure 4.15 illustrates the results using HDF5. The achieved figures are significantly lower than the figures of the disjoint pattern (1 MiB chunk size). Furthermore, independent I/O realises better results than collective I/O.

When writing independently, the 2-OST pattern performs slightly better than the 1-OST pattern; we believe that the reason lies in the increased metadata overhead when using smaller chunk sizes. The results of the disjoint pattern (illustrated in Figure 4.7) indicate that the performance behaviour is very sensitive when using such small chunk sizes. The 5- and 10-OST pattern perform worse, surely due to the increased cost of communication between clients and servers implied by the access pattern. The 1-OST

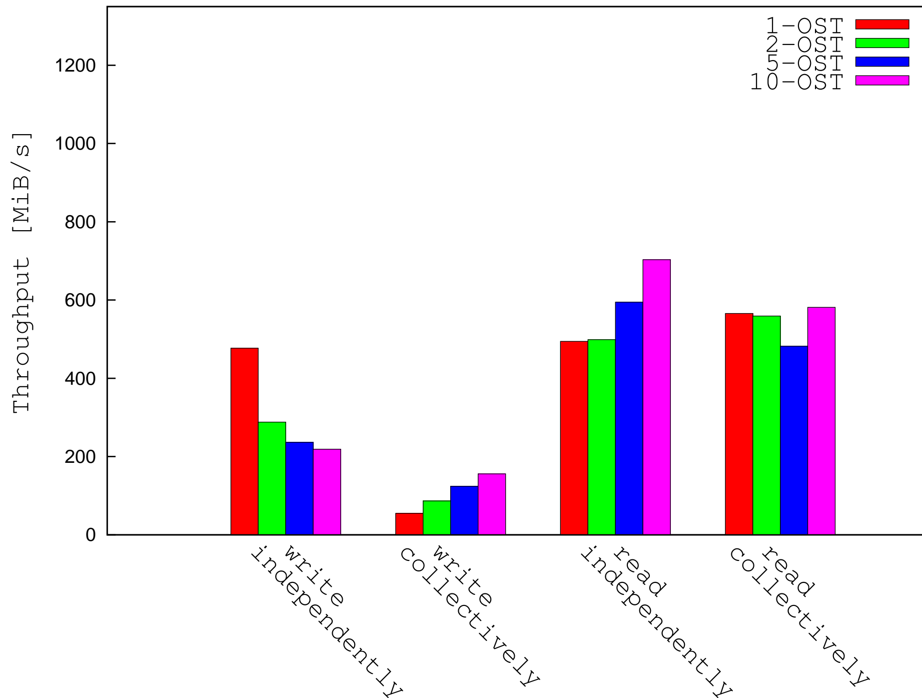

Figure 4.14.: 1- to 10-OST pattern, NetCDF-4

pattern performs best in the case of collective writes. We have the following explanation for this issue: We have inspected the locations of the chunks in the file and observed that about 85% of the transfer calls are 1-OST, which means that each client communicates with one OST without competition of other clients. It seems that the performance benefits from this lack of competition. Reading performs worst with the 1-OST pattern, due to the increased metadata overhead. This is especially expressed in the case of independent I/O. When reading collectively, the APIs perform almost equally.

Figure 4.16 shows the results using NetCDF-4. The performance among the APIs is similar to the one using HDF5. Except for the case of reading collectively with the 5-OST pattern, HDF5 performs always better than NetCDF-4. Furthermore, when writing collectively, the 1-OST pattern doesn't perform best with NetCDF-4 any more, due to the unaligned I/O which destroys the benefit mentioned above, that most of the calls were 1-OST.

4.5.3. Two-dimensional data layout

In the following test we analyse the 1-OST scenario using two-dimensional data layout with the IOR configuration stated in Table 4.4. The results are shown in figure 4.17. The results are nearly identical using HDF5 and similar when using NetCDF-4 with one dimension. We omit further tests using two-dimensional data layout, because we believe

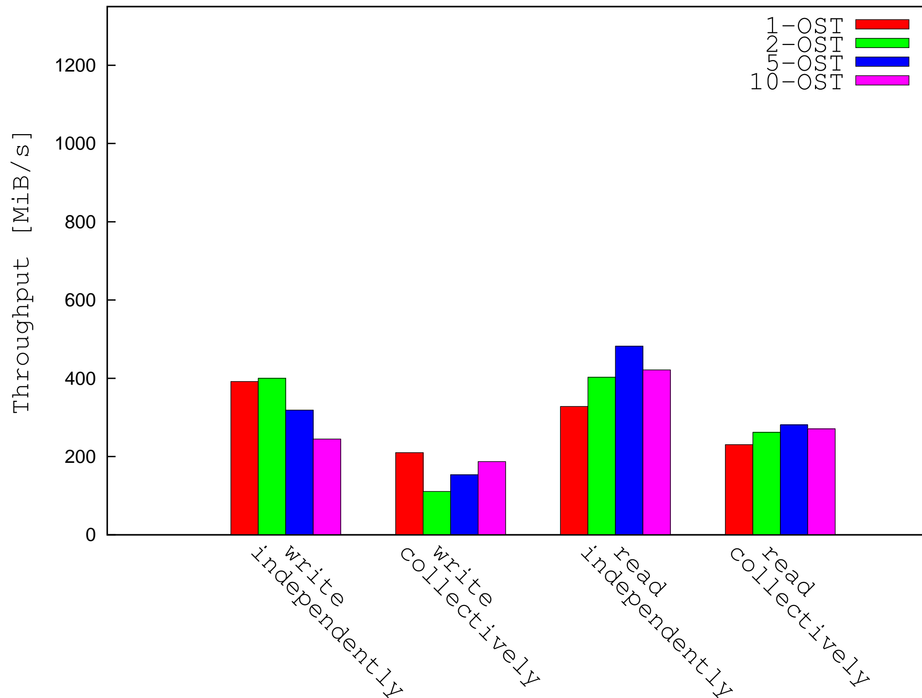

Figure 4.15.: 1- to 10-OST pattern, HDF5 with chunked I/O

that the results will be almost identical to the one-dimensional case.

Finally we analyse the performance when incrementally extending size of the dataspace (see Section 2.2). To do this, we use a two-dimensional data layout and the first dimension will be the unlimited dimension. When using HDF5, we set the initial dimension size of the first dimension to 1. As mentioned in Section 2.2, collective and chunked I/O is required in this case. We use the 1-OST pattern and set the size of the second dimension to 10 MiB, which has the consequence that the interfaces extend the filespace after each collective call. The IOR configuration is stated in Table 4.9. Figure 4.18 shows the results. NetCDF-4 performs equally to the poor figures when writing collectively using chunked I/O with the 1-OST pattern. The extension of the dataspace, which is performed after each call, has no impact on the figures. HDF5 performs much worse than using fixed-size datasets and also worse in comparison to the figures when using the 1-OST pattern which chunked I/O. It seems that the extension of the dataset slows down the performance.

4.5.4. Summary

The 1-OST pattern using POSIX or MPI-IO achieve figures which are almost the theoretical maximum. HDF5 achieves similar figures when reading and achieves higher figures as in the disjoint pattern (with 512 MiB transfer size) when writing independently.

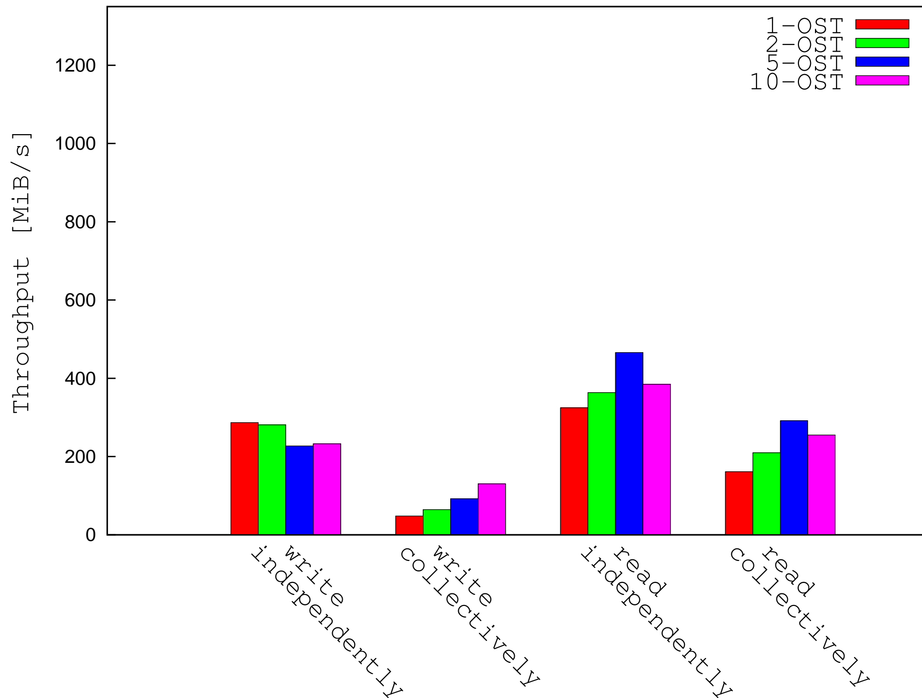

Figure 4.16.: 1- to 10-OST pattern, NetCDF-4 with chunked I/O

Writing collectively performs equal to the figures of the disjoint pattern. Collective I/O adds unnecessary synchronisation and slows down the performance. The performance of HDF5 with the 1-OST pattern benefits from large transfer sizes; especially in the case of collective I/O. Large transfer sizes result in fewer calls of the respective write/read routines and decreases the synchronisation overhead. The 2-OST pattern slows down the performance of POSIX, MPI-IO and HDF5 dramatically, due to the competition on OST and network resources.

The NetCDF-4 API achieves much lower figures than the other APIs with the 1-OST pattern. The reason lies in the lack of alignment in the API, which does not result in a 1-to-1 communication between client and server. When writing independently, NetCDF-4 achieves the highest figures with the 1-OST pattern. Writing collectively performs very poorly; the best performance is realised with the 10-OST pattern. Reading performs best with the 10-OST pattern.

Chunked I/O achieves figures with are significantly lower than when using the disjoint access pattern. Beyond that, independent I/O performs better than collective I/O. When writing independently, HDF5 and NetCDF-4 perform best with the 1- and 2-OST pattern. Writing collectively performs best with the 1-OST pattern and HDF5 and best with the 10-OST pattern with NetCDF-4. Reading realises the best figures with the 5-OST pattern.

Finally, our tests indicate that the dimensionality of the dataset does not influence the

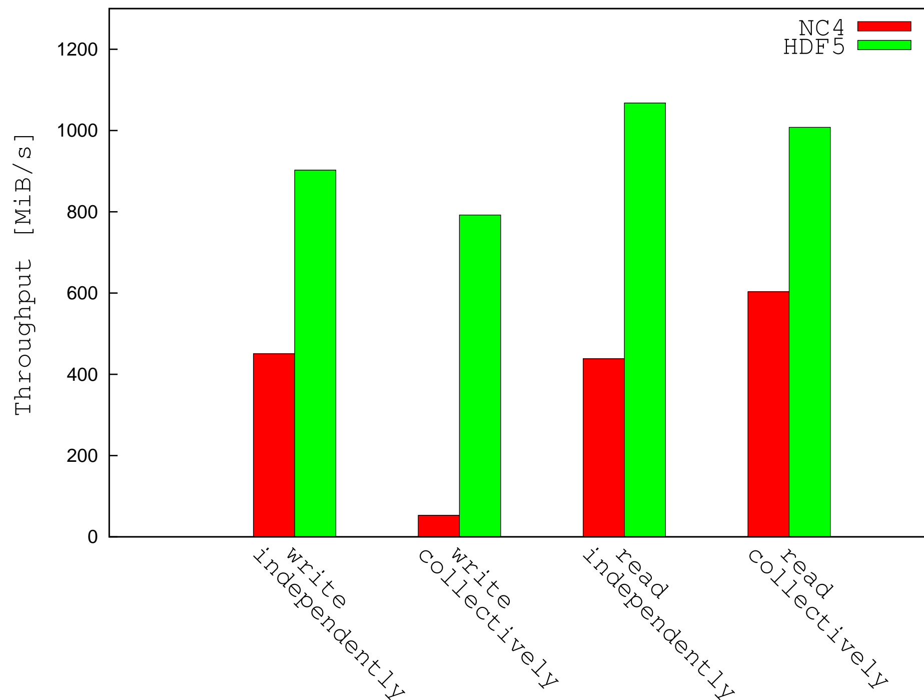

Figure 4.17.: 1-OST pattern multiple dimensions

performance behaviour. But the usage of extendible datasets slows down the performance when using HDF5.

unlimited	=	1
dimy	=	10m
segmentCount	=	20k
blockSize	=	1m
collective	=	1
chunkx	=	1
chunky	=	1m

Table 4.9.: IOR configuration - Extendible dataset

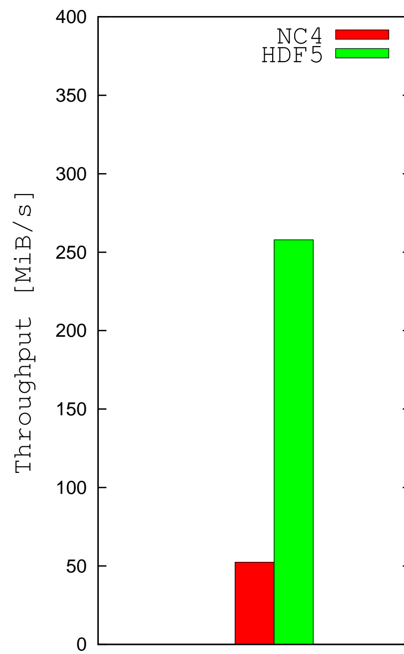

Figure 4.18.: 1-OST pattern, extending the dataspace

5. Enhancements

In this chapter enhancements to the NetCDF-4 API are presented and evaluated.

5.1. Alignment

As mentioned earlier, the NetCDF-4 API does not implement the aligning of the data objects (contiguous data section or chunks). We have added the respective function call (of the HDF5 API) and recompiled the API. We have reevaluated the disjoint and 1-OST pattern with NetCDF-4 aligned to the Lustre stripe sizes.

In our first reevaluation, we analyse the performance of NetCDF-4 using the disjoint access pattern, as stated by the IOR configuration in Table 4.3. The results are illustrated in Figure 5.1. The plot shows minimum and maximum value, because of the high standard deviation. The alignment improves the write performance of NetCDF-4, which is now almost equal to the performance of HDF5. Reading performs similar to the unaligned access.

We reevaluate the test using the 1-OST pattern, which used the IOR configuration stated in Table 4.5. We observe that the alignment of NetCDF-4 improves the performance greatly, because each client node is now really communicating with one OST node. The write performance (either independent or collective mode) is now almost equal to HDF5. Reading performs worse than HDF5, but perspicuously better than the old unaligned version of the NetCDF-4 API.

Figure 5.3 shows the reevaluation using chunked I/O and the 1-OST pattern. It shows that again the alignment improves the performance, which is significantly expressed when writing collectively. But the figures of HDF5 are not achieved, which indicates that an API overhead does exist in this case.

Summary The lack of alignment in the NetCDF-4 API is a real disadvantage regarding the performance. We have contacted Unidata concerning this issue. They have answered that "benefits of the alignment feature were apparently overlooked in the NetCDF-4 implementation" and that the feature will be implemented in a future release.

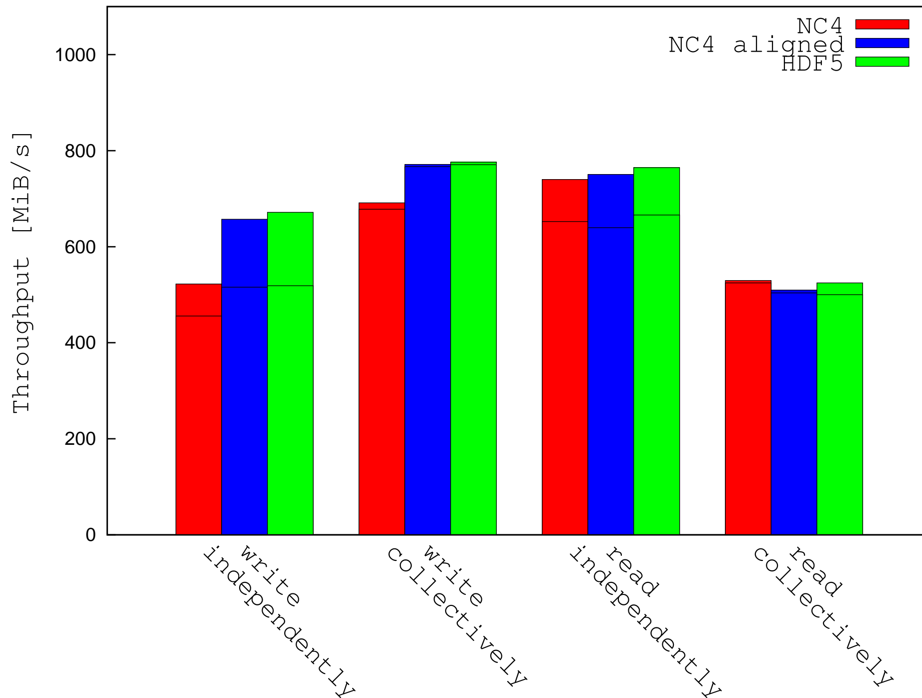

Figure 5.1.: Disjoint pattern, with NetCDF-4 aligned

Overall, the figures improve greatly when using aligned I/O. The use of the 1-OST pattern benefits the most from the alignment, as the communication is now really a 1-to-1 communication between client and server nodes.

5.2. Default chunksize using unlimited dimension

When an unlimited dimension is used in the NetCDF-4 API, the underlying storage layout must be chunked (see Section 2.2). The user can specify the sizes of each chunk dimension and default values are provided if the user does not specify a value for a dimension. The default chunk size can be provided when the NetCDF-4 API is compiled. If the value is not provided, the default chunk size equals 4 MiB. The default values for each dimension are calculated with the algorithm stated in listing 5.1. Each unlimited dimension is set to size 1, and the other dimensions in a manner that the overall chunk has the default chunk size. This algorithm is unuseable when using only unlimited dimensions and accessing large datasets, because the chunk size is too small. We have added code which checks if only one unlimited dimension is used, and in this case assigns the default chunk size even if the dimension is unlimited. We omit the multi-dimensional case, because we believe that the results will be similar. In the multi-dimensional case, one could set all dimensions to size 1 and the last varying dimension to the default chunk size. This should result in the same access pattern.

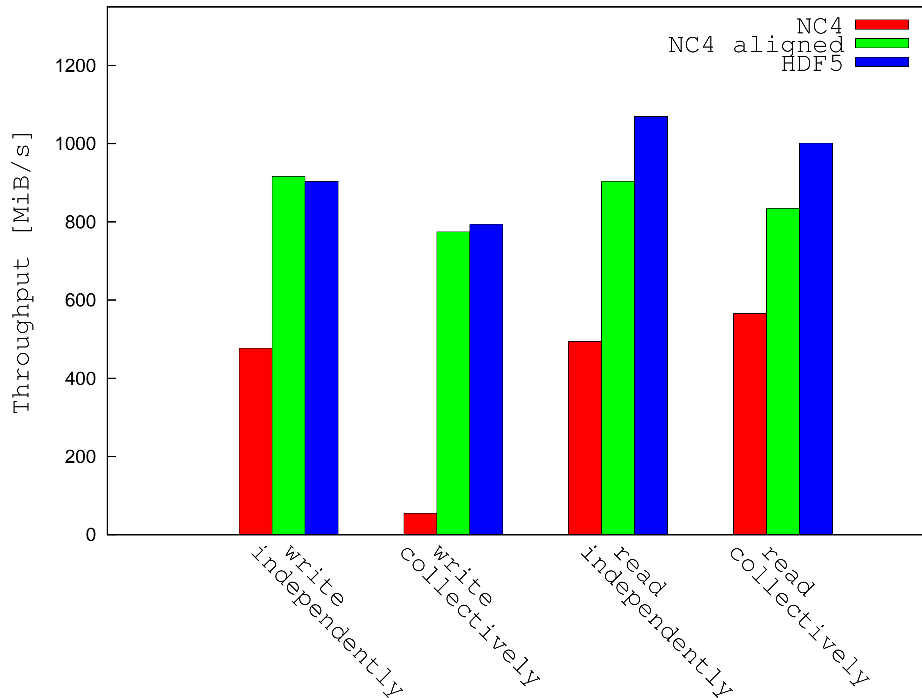

Figure 5.2.: 1-OST pattern, with NetCDF-4 aligned

```
1 if (dim->unlimited)
2    chunksize[d] = 1;
3 else
4    chunksize[d] = pow((double)DEFAULT_CHUNK_SIZE/type_size,
5                       1/(double)(var->ndims - unlimdim));
```

Listing 5.1: Algorithm calculating default chunksize

We evaluate our implementation using the alignment optimisation of section 5.1 and the disjoint and 1-OST pattern. We use the IOR configurations of Tables 4.3 and 4.5 , adding the IOR parameter *unlimited* and *collective*, to enable the unlimited dimension. Because the data rates of the unoptimised version are too small, we are just accessing 10 MiB, thus the parameter *blockSize* is set to 1 MiB in this case. In this special case, disjoint and 1-OST pattern coincide. The results are illustrated in Figure 5.4. The reader has to take notice of the logarithmic scaling of the y-axis. The data rates of the unoptimised version are unuseable, achieving one hundredth of one MiB/s when writing and about half a MiB/s when reading. The optimised version achieves, in both patterns, figures which are more than 10, 000 times better than the unoptimised version. The write performance of the disjoint pattern is similar to the contiguous case, which is illustrated in Figure 4.7. Furthermore, the disjoint pattern achieves better figures than the 1-OST pattern, especially when writing. This coincides with our results from the Evaluation Chapter.

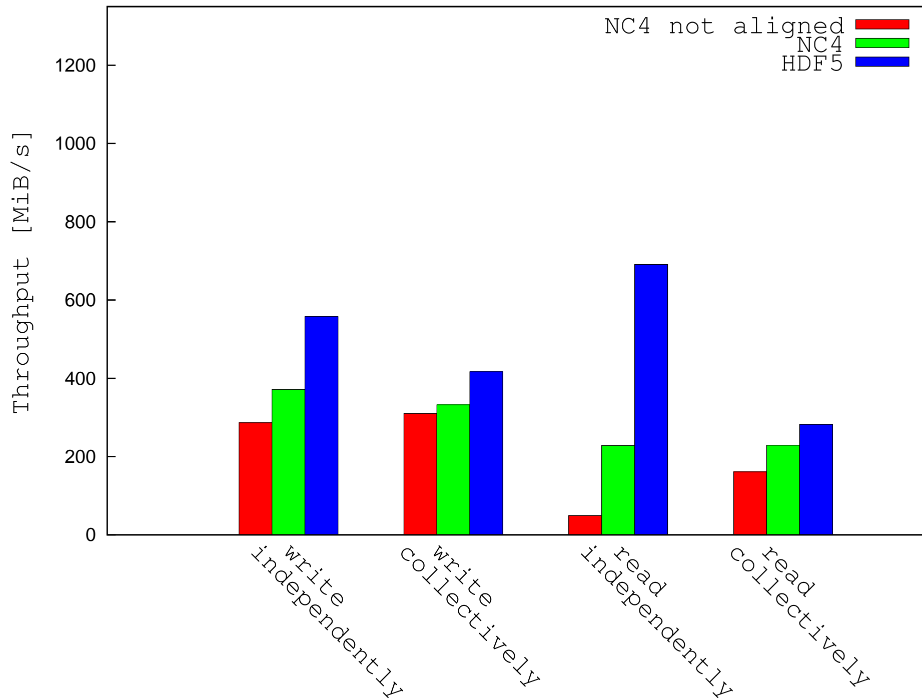

Figure 5.3.: 1-OST pattern, chunked I/O, with NetCDF-4 aligned

5.3. Internal error when using unlimited dimension

While we have run the tests of the Section 5.2, we have observed an internal error of the NetCDF-4 API. Each time the application transfers data, the length of the dimension is growing, which means that the underlying dataset has to be extended. When a dataset has to be extended, the processes must communicate with each other to find the new dimension size for each dimension, which corresponds to the maximum dimension size of all processes. To find this maximum, the NetCDF-4 API uses the MPI function Allreduce, whose signature is stated in listing 5.2.

```
1 int MPI_Allreduce(void *sendbuf, void *recvbuf, int count,
2     MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Listing 5.2: Signature of MPI_Allreduce

Allreduce performs a global reduce operation, specified by *op*, on elements of an input buffer (*sendbuf*), and stores the result in an output buffer (*recvbuf*). The elements of the buffer have type *datatype* and the size of the buffers is determined by the *count* of elements. *comm* designates the group of processes which are communicating. The NetCDF-4 API uses the maximum operation and uses MPI_INT as basic datatype, which stands for a 32-bit signed integer. But the elements of the buffer are stored as signed numbers (datatype hsize_t). Thus, the unsigned numbers are interpreted as signed numbers which leads to an error if an unsigned number is interpreted as negative

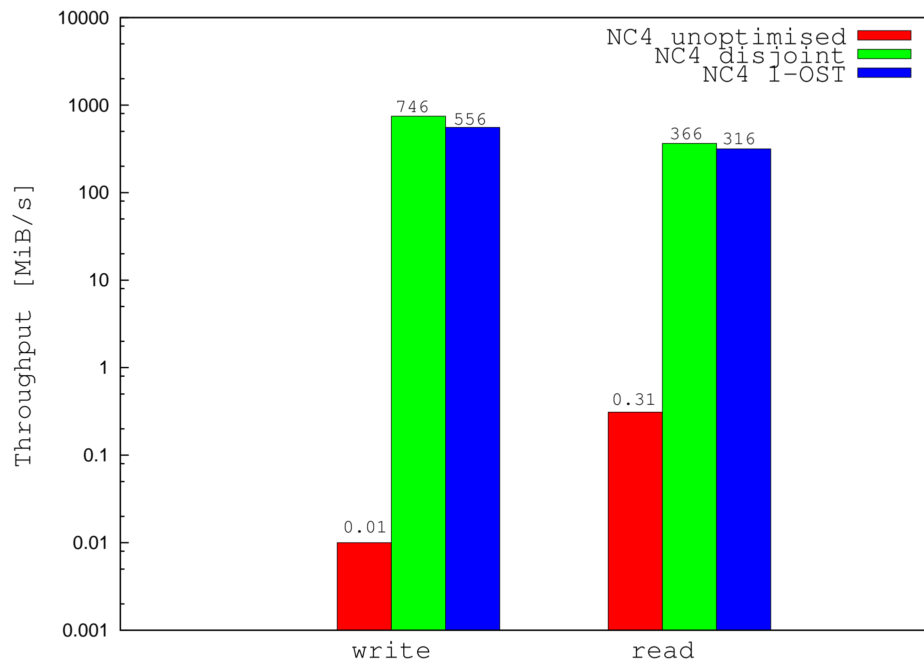

Figure 5.4.: Default chunksize using unlimited dimension

number. The correct solution would be to use MPI_Unsigned as basic datatype, which stands for an unsigned 32-bit integer. We have implemented the correct solution and started an issue on the respective GitHub repository of NetCDF-4[1]. The solution has been tested and committed to the source code by UniData.

[1]https://github.com/Unidata/netcdf-c

6. Conclusion

Summary The goal of this thesis is to analyse the performance behaviour of the HDF5 and NetCDF-4 interfaces when using parallel I/O with Lustre as underlying filesystem. Lustre splits a file up into multiple objects, called stripes, which are stored on different OSTs. These stripes are distributed among the OSTs in a round-robin fashion. HDF5 and NetCDF-4 store the data in multi-dimensional arrays. The underlying data layout can be contiguous, which means that the data is stored in one large contiguous chunk. But the data layout can also be chunked, which means that the data is broken into pieces, which are stored in independent locations. Furthermore, the HDF5 API provides the ability to align the data objects (contiguous data section or chunks) to particular address boundaries. This can be used to align the data objects to the Lustre stripes, to minimise the overhead of communication with more OSTs. Unfortunately, the NetCDF-4 API does not provide this feature.

HDF5 and NetCDF-4 use MPI-IO to perform the parallel I/O and MPI-IO uses the POSIX interfaces of the underlying operating system. Therefore, we analysed the full I/O path (POSIX, MPI-IO, HDF5 and NetCDF-4). We used the well known benchmark IOR for our evaluation and modified it to fit our analysis purposes. The original IOR software has drivers for POSIX, MPI-IO, HDF5 but not NetCDF-4. Instead, PNETCDF, a parallel API for NetCDF files with a version less than 4 is provided. We implemented a driver for NetCDF-4. Furthermore, we implemented chunking, two-dimensional data layout and support for unlimited dimension for the HDF5 and NetCDF-4 driver.

We distinguished two general access pattern for our evaluation in Chapter 4: disjoint and interleaved access. In the first pattern, each client process accesses a large contiguous region and in the second pattern a large non-contiguous region. We analysed the performance of contiguous and chunked layout; using a one-dimensional data shape. Furthermore, the impact of the usage of multiple dimensions was tested with an analysis of a two-dimensional data layout.

The figures of the disjoint access are far away from the theoretical maximum performance, because, due to the large contiguous region accessed by each client, each client has to communicate with all server nodes, which leads to high contention on network and OST resources. We observed a high variation of the achieved performance in several cases when using independent I/O. Beyond that, the write performance is very poor as some servers are not able to respond to the write requests of the clients. The results using the contiguous layout show that the NetCDF-4 API (whether reading or writing) and

the HDF5 API benefit from large transfer sizes when reading . With very large transfer sizes (512 MiB), reading does even perform better than writing. The performance of the HDF5 API is better than that of the NetCDF-4 API, when the I/O accesses are aligned to the Lustre stripe sizes. But, if HDF5 uses unaligned I/O accesses, the performance is almost equal. This indicates that the NetCDF-4 API adds almost no overhead with this access pattern. Chunked I/O benefits from large chunk sizes. The metadata overhead when using small chunks slows down the performance dramatically. The results are equal to the contiguous case if large chunk sizes are used in combination with a large transfer size (512 MiB). Finally, the dimensionality of the data layout appears to have no real impact on the performance behaviour.

In the interleaved access case, we analysed different x-OST patterns, which means that clients logically communicate with x OSTs. The 1-OST pattern with POSIX or MPI-IO achieves almost the theoretical maximum. HDF5 (with contiguous layout, aligned to the Lustre stripes) outperforms the figures achieved from the disjoint access. The reason is the lack of competition on network and OST resources. HDF5 performs better with independent I/O, as the synchronisation overhead induced by collective I/O slows down the performance. Furthermore, the performance of HDF5 benefits from large transfer sizes. But the write performance of HDF5 is lower than that of POSIX and MPI-IO, indicating that an API overhead does exist. The 2-OST pattern slows down the performance of POSIX, MPI-IO and HDF5 significantly, because of the competition on OST and network resources.

The NetCDF-4 API performs perspicuously worse than the other APIs with the 1-OST pattern. The lack of alignment in the API does not result in a 1-to-1 communication between client and server, inducing more competition on OST and network resources. The achieved figures with chunked layout are significantly lower than when using the disjoint access pattern. Beyond that, independent I/O performs better than collective I/O with chunking. Finally, like in the disjoint access pattern, our tests indicate that the dimensionality of the dataset does not influence the performance behaviour. But the usage of extendible datasets slows down the performance when using HDF5.

We presented some enhancements to the NetCDF-4 API implementation in Chapter 5. As mentioned above, the NetCDF-4 API lacks the alignment feature. We implemented the alignment to the Lustre stripes and reevaluated the disjoint and 1-OST scenario. The figures improve greatly with the aligned version, especially with the 1-OST pattern. We contacted UniData, the organisation which is responsible for the source code of NetCDF-4, and they will build this feature in a future release. Beyond that, we implemented another version of the routine which assigns default chunk sizes in NetCDF-4. We observe figures which are $10,000$ times better than the unoptimised version. Furthermore, while developing the version, we have observed an error in the API, using a false datatype. We provided the correct solution to the respective GitHub repository. The solution has been incorporated in the source code.

Best practices Our results allow us to provide best practices concerning the usage of parallel I/O with the HDF5 and NetCDF-4 APIs in order to achieve high throughput. Surely, our test environment does not equal the environment of a large HPC cluster, but we are sure that our results can be transferred to larger environments. HDF5 performs best with contiguous layout, if the 1-OST pattern is used with I/O accesses aligned to the Lustre stripes. Furthermore, independent I/O should be used in this case, as the synchronisation implied by collective I/O is unnecessary and slows down the performance. A large transfer size does also increase the performance.

Chunked I/O achieves lower performance; but in some cases (for example if the usage of compression is desired) it is required. The best performance with chunking is realised with the disjoint pattern in combination with large chunk sizes, in order to decrease the metadata overhead. Furthermore, collective I/O should be used as we have observed figures with high variation using independent I/O with this access pattern. Finally, we believe that chunking does benefit from large transfer sizes, too. Due to the lack of alignment in the NetCDF-4 API, the disjoint pattern with large transfer sizes is better suited as an interleaved pattern, if high performance is desired. Chunked I/O with NetCDF-4 does also perform best with the disjoint pattern.

Furthermore, our tests indicate that the dimensionality of the data layout has no impact on the performance behaviour. We believe that the underlying access pattern (disjoint or interleaved) is the key factor regarding performance. For the reason of separation of concern, an access pattern performance optimisation, for example like a redistribution of the data to achieve a 1-OST pattern, should be put in the MPI-IO layer. Some of such optimisations have been presented in Chapter 3, for example the user-level library Y-Lib or the static-cycling and group-cycling method. Finally, the usage of unlimited dimensions or extendible datasets should be avoided if maximal performance is desired, because the extension of the filespace slows down the performance.

Further work We used Lustre as underlying parallel filesystem. Evaluations with GPFS, which is also very common in the HPC community, could be interesting, too. The metadata overhead when using chunked layout slows down the performance dramatically in the case of small chunk sizes. The performance of HPC applications using chunked I/O would benefit if deficiencies regarding the handling of the metadata could be revealed and removed. Beyond that, a comparison of the performance of PNETCDF and NetCDF-4 might induce the need for an investigation of PNETCDF's source code, in order to find hints for parallel I/O optimisations in the NetCDF-4 API.

Bibliography

[BBC+98] Jason Barkes, Marcelo R Barrios, Francis Cougard, Paul G Crumley, Didac Marin, Hari Reddy, and Theeraphong Thitayanun. Gpfs: a parallel file system. *IBM International Technical Support Organization*, 1998.

[BM02] Rudolf Bayer and Edward McCreight. *Organization and maintenance of large ordered indexes*. Springer, 2002.

[BM06] Heiko Bauke and Stephan Mertens. *Cluster computing: praktische einführung in das hochleistungsrechnen auf linux-clustern*. Springer, 2006.

[BZ02] Peter J Braam and R Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[DL08] Phillip Dickens and Jeremy Logan. Towards a high performance implementation of mpi-io on the lustre file system. In *On the Move to Meaningful Internet Systems: OTM 2008*, pages 870–885. Springer, 2008.

[DL10] Phillip M Dickens and Jeremy Logan. A high performance implementation of mpi-io for a lustre file system environment. *Concurrency and Computation: Practice and Experience*, 22(11):1433–1449, 2010.

[DLP03] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[For09] Message Passing Interface Forum. `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`, 2009.

[G+00] HDF Group et al. Hierarchical data format version 5. *Software package, URL http://www. hdfgroup. org/HDF5*, 2000.

[Gro13] HDF Group. `http://www.hdfgroup.org/HDF5/doc/UG/UG_frame17SpecialTopics.html`, 2013. [Accessed December 2013].

[How12] Mark Howison. Tuning hdf5 for lustre file systems. 2012.

[iora] Ior user guide. `https://asc.llnl.gov/sequoia/benchmarks/IOR_User_`

Guide.pdf. [Accessed February 2014].

[IORb] IOR. https://github.com/chaos/ior.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. In *Tools for High Performance Computing*, pages 139–155. Springer, 2008.

[Kor11] Sandeep Koranne. Hierarchical data format 5: Hdf5. In *Handbook of Open Source Tools*, pages 191–200. Springer, 2011.

[Kun13] Julian Kunkel. *Simulation of Parallel Programs on Application and System Level*. Phd thesis, Universität Hamburg, 07 2013.

[Lab] Argonne National Laboratory. http://www.mcs.anl.gov/research/projects/romio/.

[LC08] Wei-keng Liao and Alok Choudhary. Dynamically adapting file domain partitioning methods for collective i/o based on underlying parallel file system locking protocols. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[LLC+03] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 39–39. IEEE, 2003.

[MPI] http://www.mpich.org/.

[NCO] http://nco.sourceforge.net/.

[NLC08] Arifa Nisar, Wei-keng Liao, and Alok Choudhary. Scaling parallel i/o performance through i/o delegate and caching system. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[OI14] Oracle and Intel. http://build.whamcloud.com/job/lustre-manual/lastSuccessfulBuild/artifact/lustre_manual.xhtml, 2014. [Accessed January 2014].

[Opea] http://www.open-mpi.de/.

[Opeb] OpenSFS. http://www.opensfs.org/press-releases/

lustre-file-system-version-2-5-released/.

[PNE] https://trac.mcs.anl.gov/projects/parallel-netcdf.

[RDE+10] Russ Rew, Glenn Davis, Steve Emmerson, Harvey Davies, and Ed Hartnett. the netcdf users guide-data model, programming interfaces, and format for self-describing, portable data-netcdf version 4.1. *Unidata Program Center*, 2010.

[ROM] http://www.mcs.anl.gov/research/projects/romio/doc/users-guide/node6.html#sec:hints.

[SAS08] Hongzhang Shan, Katie Antypas, and John Shalf. Characterizing and predicting the i/o performance of hpc applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 42. IEEE Press, 2008.

[SOW+95] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: the complete reference*. MIT press, 1995.

[SS07] Hongzhang Shan and John Shalf. Using ior to analyze the i/o performance for hpc platforms. 2007.

[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.

[Top13] Top500. http://www.top500.org, 2013. [Accessed November 2013].

[Wik13] Wikipedia. http://en.wikipedia.org/wiki/List_of_device_bandwidths#Local_area%_networks, 2013. [Accessed November 2013].

[YVCJ07] Weikuan Yu, Jeffrey Vetter, R Shane Canon, and Song Jiang. Exploiting lustre file joining for effective collective io. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 267–274. IEEE, 2007.

List of Figures

1.1 Possible architecture of an HPC system . . . 7
1.2 Non-contiguous access pattern . . . 9
1.3 I/O layering . . . 9

2.1 Two-Phase I/O . . . 13
2.2 Contiguous dataset . . . 14
2.3 Chunked dataset . . . 15
2.4 2×2 chunks overlaying a 3×3 dataspace . . . 15
2.5 Architecture of Lustre . . . 19
2.6 IOR shared file . . . 20
2.7 Vampir . . . 22

3.1 all-to-all pattern . . . 23
3.2 2-OST pattern . . . 25

4.1 Disjoint pattern . . . 28
4.2 Interleaved pattern . . . 28
4.3 Disjoint pattern . . . 34
4.4 Some clients defer writes . . . 35
4.5 Varying transfer size . . . 37
4.6 Comparing NetCDF-4 with HDF5 unaligned . . . 38
4.7 Chunked layout . . . 39
4.8 Disjoint pattern multiple dimensions . . . 40
4.9 1-OST pattern . . . 42
4.10 1-OST pattern, comparing transfer size . . . 43
4.11 2-OST pattern . . . 44
4.12 Interleaved access . . . 45
4.13 2-OST pattern, comparing transfer size . . . 45
4.14 1- to 10-OST pattern, NetCDF-4 . . . 47
4.15 1- to 10-OST pattern, HDF5 with chunked I/O . . . 48
4.16 1- to 10-OST pattern, NetCDF-4 with chunked I/O . . . 49
4.17 1-OST pattern multiple dimensions . . . 50
4.18 1-OST pattern, extending the dataspace . . . 51

5.1 Disjoint pattern, with NetCDF-4 aligned . . . 53
5.2 1-OST pattern, with NetCDF-4 aligned . . . 54

5.3 1-OST pattern, chunked I/O, with NetCDF-4 aligned 55
5.4 Default chunksize using unlimited dimension 56

List of Tables

4.1 Software versions 29
4.2 Basic IOR configuration 33
4.3 IOR configuration - disjoint pattern 34
4.4 IOR configuration - Two dimensions 38
4.5 IOR configuration - 1-OST pattern 41
4.6 IOR configuration - 2-OST pattern 44
4.7 IOR configuration - 5-OST pattern 46
4.8 IOR configuration - 10-OST pattern 46
4.9 IOR configuration - Extendible dataset 51

A.1 Results of experiments of Figure 4.3 68
A.2 Results of experiments of Figure 4.5 68
A.3 Results of experiments of Figure 4.6 69
A.4 Results of experiments of Figure 4.7 71
A.5 Results of experiments of Figure 4.9 72
A.6 Results of experiments of Figure 4.10 73
A.7 Results of experiments of Figure 4.11 73
A.8 Results of experiments of Figure 4.13 74
A.9 Results of experiments of Figure 4.14 74
A.10 Results of experiments of Figure 4.15 75
A.11 Results of experiments of Figure 4.16 75
A.12 Results of experiments of Figure 4.8 76
A.13 Results of experiments of Figure 4.17 76
A.14 Results of experiments of Figure 4.18 76
A.15 Results of experiments of Figure 5.2 77
A.16 Results of experiments of Figure 5.3 77
A.17 Results of experiments of Figure 5.1 78
A.18 Results of experiments of Figure 5.4 78

Listings

4.1 aiori struct . . . 30
4.2 NC4 data transfer algorithm . . . 31

5.1 Algorithm calculating default chunksize . . . 54
5.2 Signature of MPI_Allreduce . . . 55

Appendices

A. Experimental results

Table A.1.: Results of experiments of Figure 4.3

API	I/O	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
POSIX	independent	write	785.00	780.12	782.59	1.99
POSIX	independent	read	794.39	783.17	789.34	4.65
MPI-IO	independent	write	707.19	618.79	652.83	38.85
MPI-IO	independent	read	757.31	666.85	709.93	37.06
MPI-IO	collective	write	732.30	519.71	591.28	99.73
MPI-IO	collective	read	738.59	667.49	692.33	32.74
HDF5	independent	write	671.65	518.69	580.62	65.75
HDF5	independent	read	764.64	665.88	706.54	42.16
HDF5	collective	write	776.06	770.80	772.97	2.24
HDF5	collective	read	524.46	500.16	513.52	10.07
NC4	independent	write	522.42	455.63	498.40	30.32
NC4	independent	read	739.94	652.35	694.14	35.87
NC4	collective	write	691.28	677.96	684.47	5.44
NC4	collective	read	529.44	524.49	526.34	2.21

Table A.2.: Results of experiments of Figure 4.5

API	Transfer Size	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	128 KiB	write	486.50	475.12	481.72	4.82
	128 KiB	read	220.20	218.36	219.52	0.83
	256 KiB	write	609.29	602.58	607.02	3.14
	256 KiB	read	297.68	294.56	295.73	1.38
	512 KiB	write	804.42	802.73	803.39	0.74
	512 KiB	read	404.43	392.54	399.64	5.12
	1 MiB	write	800.45	794.79	797.79	2.32
	1 MiB	read	533.71	515.24	527.24	8.49
	2 MiB	write	801.05	798.47	800.11	1.16
	2 MiB	read	638.63	624.89	630.24	6.00
	4 MiB	write	805.63	795.25	801.07	4.33
	4 MiB	read	695.83	683.40	689.42	5.08
	8 MiB	write	805.63	802.16	803.95	1.42
	8 MiB	read	708.99	704.46	707.37	2.06

Mode	Transfer Size	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
	16 MiB	write	803.11	801.89	802.60	0.52
	16 MiB	read	733.40	726.26	728.93	3.18
	32 MiB	write	811.59	804.42	807.64	2.97
	32 MiB	read	736.39	727.10	730.36	4.26
	64 MiB	write	803.01	796.18	800.63	3.15
	64 MiB	read	790.70	784.58	787.91	2.53
	128 MiB	write	801.43	774.03	791.85	12.61
	128 MiB	read	854.94	849.09	852.84	2.66
	256 MiB	write	801.01	791.86	796.88	3.79
	256 MiB	read	860.60	842.57	852.31	7.43
	512 MiB	write	800.05	792.07	796.98	3.51
	512 MiB	read	867.73	854.60	859.46	5.88
NC4	128 KiB	write	227.71	221.76	225.03	2.47
	128 KiB	read	219.28	217.20	218.26	0.85
	256 KiB	write	442.76	437.77	439.51	2.30
	256 KiB	read	295.58	294.18	294.89	0.57
	512 KiB	write	477.31	460.21	466.27	7.82
	512 KiB	read	393.72	393.60	393.67	0.05
	1 MiB	write	677.89	665.82	670.22	5.45
	1 MiB	read	536.65	516.45	529.01	8.95
	2 MiB	write	699.76	678.31	690.11	8.89
	2 MiB	read	647.48	640.44	644.74	3.08
	4 MiB	write	779.12	772.41	776.59	2.98
	4 MiB	read	704.08	696.04	698.86	3.70
	8 MiB	write	788.37	783.51	786.11	2.00
	8 MiB	read	732.21	723.94	728.02	3.38
	16 MiB	write	790.71	788.78	789.60	0.81
	16 MiB	read	764.40	746.25	753.09	8.06
	32 MiB	write	801.63	796.82	799.66	2.06
	32 MiB	read	741.15	734.38	737.40	2.81
	64 MiB	write	805.39	791.89	799.83	5.76
	64 MiB	read	761.38	753.94	758.45	3.24
	128 MiB	write	805.38	765.58	786.40	16.30
	128 MiB	read	820.16	813.49	816.14	2.89
	256 MiB	write	805.74	786.18	798.27	8.63
	256 MiB	read	829.78	827.83	828.74	0.80
	512 MiB	write	800.45	753.10	780.15	19.92
	512 MiB	read	831.22	824.84	827.68	2.65

Table A.3.: Results of experiments of Figure 4.6

Mode	Transfer Size	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
Aligned	128 KiB	write	486.50	475.12	481.72	4.82
	128 KiB	read	220.20	218.36	219.52	0.83

	256 KiB	write	609.29	602.58	607.02	3.14
	256 KiB	read	297.68	294.56	295.73	1.38
	512 KiB	write	804.42	802.73	803.39	0.74
	512 KiB	read	404.43	392.54	399.64	5.12
	1 MiB	write	800.45	794.79	797.79	2.32
	1 MiB	read	533.71	515.24	527.24	8.49
	2 MiB	write	801.05	798.47	800.11	1.16
	2 MiB	read	638.63	624.89	630.24	6.00
	4 MiB	write	805.63	795.25	801.07	4.33
	4 MiB	read	695.83	683.40	689.42	5.08
	8 MiB	write	805.63	802.16	803.95	1.42
	8 MiB	read	708.99	704.46	707.37	2.06
	16 MiB	write	803.11	801.89	802.60	0.52
	16 MiB	read	733.40	726.26	728.93	3.18
	32 MiB	write	811.59	804.42	807.64	2.97
	32 MiB	read	736.39	727.10	730.36	4.26
	64 MiB	write	803.01	796.18	800.63	3.15
	64 MiB	read	790.70	784.58	787.91	2.53
	128 MiB	write	801.43	774.03	791.85	12.61
	128 MiB	read	854.94	849.09	852.84	2.66
	256 MiB	write	801.01	791.86	796.88	3.79
	256 MiB	read	860.60	842.57	852.31	7.43
	512 MiB	write	800.05	792.07	796.98	3.51
	512 MiB	read	867.73	854.60	859.46	5.88
Unaligned	128 KiB	write	232.90	226.54	228.85	2.88
	128 KiB	read	221.07	216.25	219.25	2.14
	256 KiB	write	432.67	413.44	425.14	8.38
	256 KiB	read	297.68	295.08	296.74	1.18
	512 KiB	write	507.79	500.44	504.17	3.00
	512 KiB	read	392.23	389.03	391.02	1.42
	1 MiB	write	652.71	639.11	645.34	5.61
	1 MiB	read	535.65	528.95	533.32	3.09
	2 MiB	write	688.61	676.51	683.53	5.13
	2 MiB	read	654.91	648.00	651.80	2.86
	4 MiB	write	784.15	775.73	779.56	4.26
	4 MiB	read	725.37	712.51	719.17	6.44
	8 MiB	write	784.59	778.73	781.61	2.39
	8 MiB	read	763.58	752.61	757.16	4.67
	16 MiB	write	793.98	780.69	788.08	5.53
	16 MiB	read	782.06	779.13	780.66	1.20
	32 MiB	write	799.77	778.70	792.63	9.85
	32 MiB	read	761.94	754.24	759.03	3.42
	64 MiB	write	803.61	770.61	788.67	13.65

	64 MiB	read	801.66	781.13	790.25	8.54
	128 MiB	write	799.31	794.62	797.66	2.15
	128 MiB	read	853.85	847.20	849.71	2.95
	256 MiB	write	798.16	794.10	795.63	1.80
	256 MiB	read	860.07	858.05	859.11	0.83
	512 MiB	write	793.46	722.44	762.21	29.61
	512 MiB	read	868.93	858.34	863.95	4.34

Table A.4.: Results of experiments of Figure 4.7

API	Chunk Size	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	128 KiB	write	384.23	381.27	382.46	1.28
		read	522.38	514.18	517.50	3.53
	256 KiB	write	520.55	515.88	518.12	1.91
		read	564.47	555.54	560.06	3.65
	512 KiB	write	636.91	634.00	635.00	1.35
		read	679.52	666.50	673.51	5.37
	1 MiB	write	710.25	694.83	704.85	7.09
		read	752.29	732.81	739.92	8.78
	2 MiB	write	748.80	717.23	729.34	13.90
		read	810.60	800.66	806.90	4.44
	4 MiB	write	781.08	774.64	777.40	2.70
		read	813.33	802.34	807.89	4.49
	8 MiB	write	778.77	769.87	773.26	3.93
		read	834.93	817.22	825.52	7.27
	16 MiB	write	784.68	763.48	775.20	8.80
		read	846.86	830.55	836.17	7.56
	32 MiB	write	797.90	780.24	791.25	7.84
		read	853.77	839.14	847.77	6.26
	64 MiB	write	787.73	774.19	783.20	6.37
		read	863.63	849.96	855.52	5.86
	128 MiB	write	800.28	772.56	784.06	11.80
		read	868.24	855.33	860.26	5.69
	256 MiB	write	786.66	782.92	784.61	1.55
		read	853.68	848.35	850.49	2.30
	512 MiB	write	785.76	777.81	781.18	3.36
		read	865.43	846.02	855.93	7.93
NC4	128 KiB	write	297.05	293.17	294.88	1.62
		read	505.08	495.32	501.09	4.18
	256 KiB	write	452.72	448.90	450.63	1.58
		read	544.09	536.97	539.71	3.12
	512 KiB	write	560.47	552.71	555.91	3.31
		read	652.99	644.79	648.51	3.39
	1 MiB	write	628.11	617.19	622.72	4.46

	read	685.33	677.67	682.62	3.51
2 MiB	write	716.81	711.55	713.78	2.22
	read	767.17	746.50	758.30	8.69
4 MiB	write	711.05	691.43	702.75	8.29
	read	786.52	772.27	777.45	6.44
8 MiB	write	778.93	777.99	778.43	0.39
	read	801.09	798.37	799.47	1.17
16 MiB	write	768.91	753.35	762.65	6.71
	read	806.88	788.86	798.81	7.47
32 MiB	write	790.13	774.65	781.16	6.56
	read	819.73	802.51	811.74	7.09
64 MiB	write	775.23	766.41	770.50	3.63
	read	821.66	817.82	819.90	1.59
128 MiB	write	784.84	767.15	773.69	7.92
	read	833.45	824.59	828.32	3.75
256 MiB	write	789.73	774.56	783.99	6.72
	read	827.68	824.78	826.68	1.34
512 MiB	write	796.96	791.69	794.88	2.29
	read	827.90	819.74	824.36	3.42

Table A.5.: Results of experiments of Figure 4.9

API	I/O	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
POSIX	independent	write	1075.61	1071.89	1073.73	1.52
POSIX	independent	read	1069.26	1062.97	1066.23	2.58
MPI-IO	independent	write	1076.32	1070.43	1073.93	2.53
MPI-IO	independent	read	1070.59	1069.89	1070.19	0.29
HDF5	independent	write	905.69	900.70	903.80	2.20
HDF5	independent	read	1071.97	1068.53	1069.87	1.50
HDF5	collective	write	797.82	789.92	792.61	3.68
HDF5	collective	read	1007.11	996.25	1001.44	4.45
NC4	independent	write	504.69	453.80	477.15	20.99
NC4	independent	read	521.14	480.28	494.52	18.84
NC4	collective	write	55.13	54.58	54.8	0.29
NC4	collective	read	574.35	549.36	565.69	11.56

Table A.6.: Results of experiments of Figure 4.10

API	I/O	Transfer Size	Op.	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
MPI-IO	ind.	1 MiB	write	1076.32	1070.43	1073.93	2.53
MPI-IO	ind.		read	1070.59	1069.89	1070.19	0.29
HDF5	ind.		write	905.69	900.70	903.80	2.20
HDF5	ind.		read	1071.97	1068.53	1069.87	1.50
HDF5	cll.		write	797.82	789.92	792.61	3.68
HDF5	cll.		read	1007.11	996.25	1001.44	4.45
MPI-IO	ind.	512 MiB	write	1079.69	1074.03	1077.63	3.13
MPI-IO	ind.		read	1072.16	1067.99	1070.60	2.28
HDF5	ind.		write	960.91	958.52	959.82	1.21
HDF5	ind.		read	1072.20	1065.30	1067.68	3.92
HDF5	cll.		write	958.66	956.65	957.90	1.09
HDF5	cll.		read	1067.89	1064.02	1066.17	1.97

Table A.7.: Results of experiments of Figure 4.11

API	I/O	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
POSIX	independent	write	511.71	487.03	499.93	10.11
POSIX	independent	read	383.58	352.01	368.18	12.90
MPI-IO	independent	write	530.08	486.89	510.99	17.98
MPI-IO	independent	read	468.45	339.68	419.62	56.99
HDF5	independent	write	505.66	496.64	502.51	4.15
HDF5	independent	read	800.95	359.44	522.37	197.93
HDF5	collective	write	122.42	121.85	122.06	0.26
HDF5	collective	read	780.35	765.29	772.28	6.20
NC4	independent	write	295.48	280.82	288.24	5.99
NC4	independent	read	504.46	494.93	498.75	4.11
NC4	collective	write	86.83	86.12	86.49	0.29
NC4	collective	read	562.88	556.58	559.13	2.71

Table A.8.: Results of experiments of Figure 4.13

API	I/O	Transfer Size	Op.	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
MPI-IO	ind.	1 MiB	write	530.08	486.89	510.99	17.98
MPI-IO	ind.		read	468.45	339.68	419.62	56.99
HDF5	ind.		write	505.66	496.64	502.51	4.15
HDF5	ind.		read	800.95	359.44	522.37	197.93
HDF5	cll.		write	122.42	121.85	122.06	0.26
HDF5	cll.		read	780.35	765.29	772.28	6.20
MPI-IO	ind.	512 MiB	write	306.58	276.96	293.74	15.20
MPI-IO	ind.		read	749.51	701.27	725.12	24.12
HDF5	ind.		write	517.56	508.55	511.77	5.03
HDF5	ind.		read	505.03	433.06	474.42	37.17
HDF5	cll.		write	313.26	298.43	304.47	7.79
HDF5	cll.		read	705.80	686.77	695.43	9.63

Table A.9.: Results of experiments of Figure 4.14

I/O	OST pattern	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
independent	1	write	504.69	453.80	477.15	20.99
independent		read	521.14	480.28	494.52	18.84
collective		write	55.13	54.58	54.8	0.29
collective		read	574.35	549.36	565.69	11.56
independent	2	write	295.48	280.82	288.24	5.99
independent		read	504.46	494.93	498.75	4.11
collective		write	86.83	86.12	86.49	0.29
collective		read	562.88	556.58	559.13	2.71
independent	5	write	244.26	231.51	236.31	5.66
independent		read	612.76	577.14	594.80	14.54
collective		write	124.99	122.39	124.08	1.20
collective		read	489.68	475.65	482.11	5.78
independent	10	write	229.42	212.99	218.84	7.50
independent		read	707.87	699.18	702.97	3.64
collective		write	161.94	151.52	155.86	4.43
collective		read	588.81	571.19	581.30	7.43

Table A.10.: Results of experiments of Figure 4.15

I/O	OST pattern	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
independent	1	write	393.90	389.47	391.91	1.84
independent		read	329.01	327.55	328.10	0.65
collective		write	211.19	209.10	210.13	0.85
collective		read	232.60	229.37	230.54	1.46
independent	2	write	401.99	398.32	399.91	1.54
independent		read	407.74	393.43	402.80	6.63
collective		write	111.22	110.48	110.81	0.31
collective		read	265.61	258.23	261.75	3.02
independent	5	write	321.21	316.49	318.87	1.93
independent		read	487.09	476.85	482.05	4.18
collective		write	153.78	153.38	153.58	0.16
collective		read	284.49	278.08	281.07	2.63
independent	10	write	255.15	236.92	244.86	7.63
independent		read	421.99	420.39	421.36	0.70
collective		write	188.08	185.93	187.04	0.88
collective		read	271.89	269.89	270.67	0.88

Table A.11.: Results of experiments of Figure 4.16

I/O	OST pattern	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
independent	1	write	293.59	282.98	286.64	4.92
independent		read	334.38	314.57	324.75	8.10
collective		write	48.67	46.85	48.00	1.00
collective		read	161.99	160.82	161.27	0.63
independent	2	write	282.79	279.92	280.95	1.31
independent		read	366.26	361.60	363.44	2.03
collective		write	64.47	64.32	64.41	0.07
collective		read	210.52	208.61	209.75	0.83
independent	5	write	242.62	219.07	227.00	11.04
independent		read	470.44	460.83	465.64	3.92
collective		write	92.96	91.64	92.08	0.62
collective		read	297.78	285.80	291.41	4.92
independent	10	write	242.33	213.90	232.52	13.17
independent		read	388.27	381.78	384.64	2.70
collective		write	142.61	123.28	130.34	8.71
collective		read	255.87	254.13	254.93	0.72

Table A.12.: Results of experiments of Figure 4.8

Api	I/O	Mode	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	ind.	unaligned	write	618.68	509.75	580.93	50.36
HDF5	ind.	unaligned	read	737.04	691.26	718.10	19.51
HDF5	ind.	aligned	write	744.82	520.99	636.60	91.53
HDF5	ind.	aligned	read	773.31	713.95	734.72	27.31
NC4	ind.	unaligned	write	569.69	497.59	522.89	33.13
NC4	ind.	unaligned	read	770.15	688.21	730.83	33.53
HDF5	cll.	unaligned	write	711.98	682.94	692.78	13.58
HDF5	cll.	unaligned	read	543.46	525.80	537.08	8.00
HDF5	cll.	aligned	write	774.93	768.62	771.58	2.59
HDF5	cll.	aligned	read	524.50	506.06	517.73	8.28
NC4	cll.	unaligned	write	680.38	668.95	675.30	4.75
NC4	cll.	unaligned	read	522.32	521.07	521.61	0.53

Table A.13.: Results of experiments of Figure 4.17

API	I/O	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	independent	write	905.34	899.92	902.42	2.23
HDF5	independent	read	1073.46	1064.07	1067.48	4.24
HDF5	collective	write	798.78	785.83	792.14	5.29
HDF5	collective	read	1024.00	989.52	1007.78	14.15
NC4	independent	write	454.88	447.69	450.42	3.18
NC4	independent	read	450.37	428.52	438.45	9.03
NC4	collective	write	52.97	52.84	52.88	0.08
NC4	collective	read	603.98	602.25	603.23	0.72

Table A.14.: Results of experiments of Figure 4.18

API	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	258.60	257.04	257.91	0.65
NC4	52.52	52.35	52.41	0.10

Table A.15.: Results of experiments of Figure 5.2

Api	I/O	Mode	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	ind.	aligned	write	905.69	900.70	903.80	2.20
HDF5	ind.	aligned	read	1071.97	1068.53	1069.87	1.50
HDF5	cll.	aligned	write	797.82	789.92	792.61	3.68
HDF5	cll.	aligned	read	1007.11	996.25	1001.44	4.45
NC4	ind.	unaligned	write	504.69	453.80	477.15	20.99
NC4	ind.	unaligned	read	521.14	480.28	494.52	18.84
NC4	cll.	unaligned	write	55.13	54.58	54.8	0.29
NC4	cll.	unaligned	read	574.35	549.36	565.69	11.56
NC4	ind.	aligned	write	922.60	905.22	916.48	7.97
NC4	ind.	aligned	read	908.55	892.40	902.51	7.19
NC4	cll.	aligned	write	780.07	770.90	774.47	4.01
NC4	cll.	aligned	read	840.07	826.94	834.68	5.61

Table A.16.: Results of experiments of Figure 5.3

Api	Mode	I/O	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	aligned	ind.	write	586.99	500.11	557.48	40.57
HDF5	aligned	ind.	read	420.18	414.10	416.55	2.62
HDF5	aligned	cll.	write	693.11	688.80	690.99	1.76
HDF5	aligned	cll.	read	282.53	281.88	282.26	0.28
NC4	unaligned	cll.	write	49.83	49.27	49.46	0.32
NC4	unaligned	cll.	read	162.45	158.87	160.85	1.82
NC4	unaligned	ind.	write	310.48	251.76	286.47	30.79
NC4	unaligned	ind.	read	318.80	304.19	310.26	6.21
NC4	aligned	cll.	write	230.95	224.72	228.50	2.72
NC4	aligned	cll.	read	231.65	227.09	229.04	1.92
NC4	aligned	ind.	write	383.58	355.33	371.66	11.94
NC4	aligned	ind.	read	332.90	330.53	332.04	1.07

Table A.17.: Results of experiments of Figure 5.1

API	I/O	Mode	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
HDF5	ind.	aligned	write	671.65	518.69	580.62	65.75
HDF5	ind.	aligned	read	764.64	665.88	706.54	42.16
HDF5	cll.	aligned	write	776.06	770.80	772.97	2.24
HDF5	cll.	aligned	read	524.46	500.16	513.52	10.07
NC4	ind.	unaligned	write	522.42	455.63	498.40	30.32
NC4	ind.	unaligned	read	739.94	652.35	694.14	35.87
NC4	cll.	unaligned	write	691.28	677.96	684.47	5.44
NC4	cll.	unaligned	read	529.44	524.49	526.34	2.21
NC4	ind.	aligned	write	657.16	515.69	565.37	64.98
NC4	ind.	aligned	read	750.53	639.59	691.27	45.60
NC4	cll.	aligned	write	771.19	766.82	768.68	1.84
NC4	cll.	aligned	read	509.44	503.75	506.88	2.36

Table A.18.: Results of experiments of Figure 5.4

Version	Mode	Operation	Max(MiB/s)	Min(MiB/s)	Mean(MiB/s)	StdDev
Old	disjoint & 1-OST	write	0.01	0.01	0.01	0.00
Old	disjoint & 1-OST	read	0.49	0.22	0.31	0.16
New	disjoint	write	747.70	743.98	745.91	1.86
New	disjoint	read	367.81	361.75	365.52	3.29
New	1-OST	write	563.28	543.27	556.04	11.09
New	1-OST	read	316.84	315.70	316.17	0.49

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 Master-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den ....