Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

MASTERTHESIS

Modeling and Performance Prediction of HDF5 data on Objectstorage

vorgelegt von

Roman Jerger

MIN-Fakultät

Fachbereich Informatik

Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik

Matrikelnummer: 5788742

Erstgutachter: Prof. Dr. Thomas Ludwig

Zweitgutachter: Dr. Julian Kunkel

CONTENTS

CHAPTER

ONE

INTRODUCTION

In this chapter, basic introduction about a popular HDF5 data format as a representative for other self-describing data formats in use is given. Another briefly discussion concern object storage technologies as an alternative to file storage. After that, the combination of this technologies - one for data access and one for data storage is motivated. Subsequently the structure of this thesis is described and a summary of this chapter is given.

1.1 Self-describing Data Formats

A self-describing data format is one that contains the information about the fields in the file. This allows an application to interpret the data stored in file without any information given from outside.

In the early years of scientific computing custom data formats have been developed for storing data. This data formats often utilize raw binary or even ASCII based formats, which had big disadvantages for using them:

- Reading and writing ASCII based formats requires additional functionality like parsing and serialisation and therefore inefficient
- Limited portability due to different representations dependent on machine architecture and used programming language:
  - Byte ordering (big endian vs. little endian)
  - Floating point representation standards (e.g. IEEE 754 vs. Cray standard)
- Reading data requires changes of implemented operational logic for every custom data organisation

The information describing the stored data is shipped within the file itself. This information allows interpretation by characterizing the contained data and optimizes access to datapoints by operations.

Standardisation of self-describing data formats also allow providing a standardized stable API as well as implementing portable libraries for efficient data access in variety of programming languages.

1.2 Hierarchical Data Format 5

HDF5 is an open source self-describing data format, which combine metadata and data itself. It is designed for flexible and efficient I/O, for high volume and complex data. Developed over 20 years HDF5 provides a large set of features highly desired in scientific environments:

- bindings for many programming languages
- portability and compatibility
- openness
- flexibility
- long term support
- tools for viewing, managing, manipulating and analyzing data

HDF5 implements a model for storing and managing large amounts of data. It includes the following components:

- File format:

  an abstract storage model for storing data.

- Data model:

  an abstract data model for storing logical structures for data management and access by an application.

- Software:

  open source libraries for accessing information stored in a HDF5 file format. This includes providing programming language API's, mapping of storage models to different storage mechanisms and providing additional tools for data management. The relationships between the different models and implementations are shown in Fig. 1.1.

Hierarchical Data Format is widely used in scientific applications for storing huge amounts of data. Due to optimizations it provides high efficient access to extremely large and complex data collections stored on backend storage systems. However, due to the growing amount of data and increasing of required precision and time limits for computing, the current bottleneck limiting the computing performance is not the data model itself, but the performance of the underlying storage technologies.

1.3 Object Storage Services

With development of cloud systems in recent years the common trend for storing of unstructured data has moved towards using object storage services for common applications. The advantages of using a object storage service for applications which are not imply high I/O load of the storage

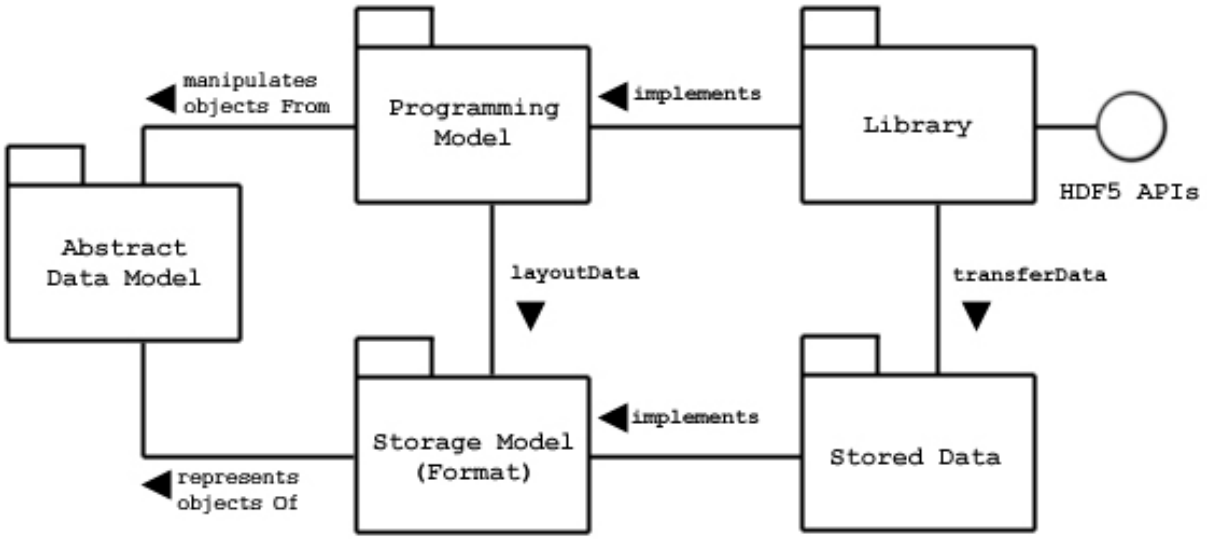

Fig. 1.1: HDF5 abstract data model

are very diverse, they reach from pure financial reasons to technological decisions. The most often named reasons are the following:

- simplicity of usage and management
- scalability
- as-a-service delivering model
- pay-per-use accounting
- interchangability through opensource/open standards
- diversity of implementations

The most interesting property in context of scientific computing is scalability. The design of object storage services allows to scale storage systems regarding to volume of stored data and I/O throughput in near linear scale. Due to the differences between the way data is organized and accessed though the file system interface and using object storage systems the tools, which actually manage access to the storage systems to have to be adopted in order to support the new storage paradigm.

Despite of the fact that storage and I/O scalability are highly desired in high performance computing, the practical advantages of accessing scientific data in the same manner as it is done by using object storage services has been underresearched.

1.4 Motivation

Scientific computing is often very I/O intensive. Scientific disciplines such as climate research, astro and nuclear physics, fluid dynamics and other have to deal with terabytes of data accessing it by thousands of servers in parallel. The data itself is stored on many servers and storage devices providing a global view on available storage resources. Scientific applications access the required data through the file system interface. However, providing a file system interface with strong consistency semantics (usually compatible with POSIX semantics) is very challenging in highly distributed environments and leads to performance forfeits.

Computational performance in critical is such environments for the overall throughput. One of the main factors which have a big influence on performance of I/O intensive computations is the performance of the underlying storage. Besides already described scaling of I/O performance many object storage systems have a additional built-in functionalities, that also could be very useful in scientific environment. These are:

- replication
- high availability
- asynchronous replication
- encryption
- deduplication
- compression
- erasure code
- cache tiering
- and other

Apart of technical considerations using object storage for storing big amounts of data also has a potential to reduce costs for maintaining a data center.

1. Object storage system are designed for utilizing commodity hardware. This has a potential to reduce the costs for IT procurement by using a more inexpensive hardware.
2. Simplified scalability could also reduse costs for personal training required for commissioning and supporting an object storage system.
3. The same reason would allow to reduce personnel expenses.

1.5 Contribution

The main purpose of this thesis is to investigate the behaviour of object storage systems in context of scientific data and applications. The key contributions of this thesis include:

- evaluation of object storage performance dependent on parameters configurable by a client application
- evaluation of object storage performance dependent on the chosen interface
- design and implementation of HDF5 plugin for storing data using object storage service
- establishing an I/O performance prediction algorithm
- evaluation of prediction algorithm and the implemented plugin

1.6 Structure

This rest of the thesis is organized as follows:

Chapter 2 introduces object storage systems. It shows the differences between file system and object storage interfaces. Additionally the relevant aspects of the HDF5 data and a programming model are introduced.

Chapter 3 deals with performance evaluation of object storage systems and interfaces.

Chapters 4 and 5 describe the design of the implementations of the plugin respectively.

The next chapter introduces the I/O performance prediction algorithm and evaluates the implemented plugin under various conditions.

The last chapter summarizes the work done and gives an overview over the future work.

1.7 Summary

This chapter has briefly introduced the self-describing file format, data model and library HDF5. It has also described object storage services and the possible advantages of using them with the HDF5 library. Subsequently the motivation and the structure of this work was provided.

CHAPTER

TWO

TECHNICAL BACKGROUND

This chapter describes the whole stack of technologies used for storing data in high performance computing and compares corresponding parts of them in a bottom up manner. In the beginning of the chapter the common architecture and the properties of object storage services are introduced. Further the differences between using file system object storage interfaces are stated out. Later the differences between different implementations of object storage systems are described and the most common object storage interface - S3 API is presented. Further in the storage hierarchy, the MPI I/O library is basically introduced. The chapter is rounded up by describing the data and programming models together with the introduction of HDF5 VOL plugin.

2.1 Object Storage

Object storage services provide an essentially different way of storing, organizing and accessing data on disk. An Object Storage platform implements a storage infrastructure to store data along with metadata as objects. Storage devices build a single storage resource for storing objects, which is represented as pool to the end user. Designed to solve storage scalability problems, object storage provides a lot of desired features for distributed storage systems:

- near linear scalability
- simplicity of management
- built-in replication
- high availability through avoidance of single point of failure
- avoiding file system limits
- simplicity of usage

Object storage systems are designed to be easily extendable. This is made possible through changing the way scalability is reached. There are two ways of scaling a system - scale up and scale out. Scaling up generally refers to purchasing and installing a more capable central control or piece of hardware. In contrast, scaling out means grouping together low performant hardware in order to collectively do the work of a much more performant one. For instance, in case of increasing

network I/O performance, scaling up would imply replacing all 1Gb/s networking hardware by 10Gb/s ones. Scaling out would mean using interface bounding.

Scaling of object storage systems is done by scaling out principal. This implies simplicity of management and cost reduction and allows a system to be extendable into peta- or even exabyte scale.

Never the less, object storage systems change the way data is accessed by applications. In contrast to file systems, object storage defines no hierarchical relations between objects. When objects are created, an identifier (a key) is created to locate the object within the pool. Using the key of the object read and write operations on data and on is metadata can be be performed, typically using HTTP protocol utilizing REST- or XML-based APIs.

Despite of the fact that every implementation of an object storage service can potentially introduce its own architecture, there are some similarities between them. Common architecture of object storage systems is depicted in Fig. 2.1.

In this architecture every controller node has knowledge about storage server, storage devices and the current state of them. With this knowledge every controller node builds a ring over storage devices in order to map object to them. Ideally this is done by uniformly distributing objects to storage devices maximizing the possible parallel throughput.

Object storage access is based on client-server architecture. A set of clients perform CRUD-operations (Create, Remove, Update, Delete) on objects and metadata using objects key. The load-balancer delegates each operation to one of the controller nodes which resides either collocated with storage devices, or deployed as a stand alone service. The controller node computes a hash over the objects key, determining on which storage node and on which storage device the object is located and delegates the operation to responsible storage node. The response is returned on the same way, or is returned directly to requester dependent of implementation.

2.2 Object Storage vs. File System Interfaces

There are differences between how an application aresses data on storage systems based on file system interface and based on object storage. File system interfaces provide a hierarchical namespace for specifying a path to the file, where a data is stored. This path consists of a set of directories names beginning with a root directory separated by a system specific separator character. The path ends with the name of the file where the data has to be read or written.

Object storage systems provide no hierarchical relationships between objects. Instead this, they provide a flat namespace, where a data is accessed by a key/value relationships, where key is name of the object and value is data itself, stored under the name of the key. The separation of the keys namespaces is made by so called buckets.

One another significant difference between file system and object storage interfaces concerns how the data is accessed. Using a file system interface, once a file is opened for read or write access, the I/O operations take place blockwise. Additionally to specifying a file descriptor, a read/write buffer

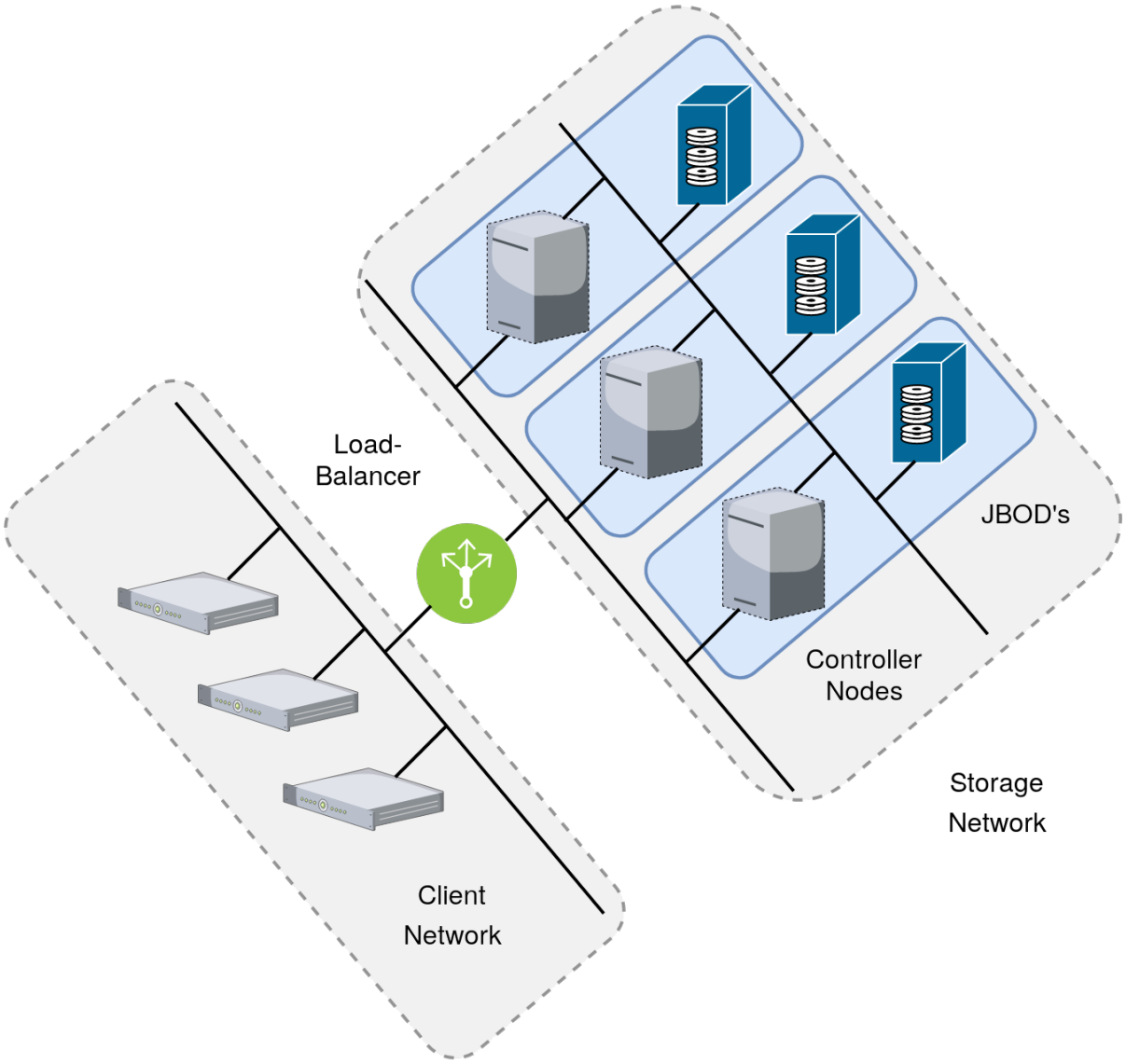

Fig. 2.1: Common architecture of object storage services

and a size of data to be read/written this requires a specifying an offset relative to the beginning of the file, where the data has to be stored.

In contrast to the file system interface objects are accessed by the name of the object (the key) and are read/written as a whole. Object storage interfaces define no partial I/O operations. The best way to illustrate this is to show the interfaces of the corresponding functions. For this purpose in Listing 2.1 and in Listing 2.2 the interfaces of I/O related operations using POSIX- and object storage interfaces are shown. In case of the object storage interface the related operations of `libs3` library are shown. Despite of the fact that the `libs3` library defines partial read operations, they are commonly not supported. Partial write operations are not supported completely.

Listing 2.1: File system related I/O operations as defined by POSIX interface

```
ssize_t pwrite(int fildes, const void *buf,
               size_t nbyte, off_t offset);
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
```

Listing 2.2: Object storage related I/O operations as defined by `libs3` library

```
void S3_put_object([...], const char *key,
                   uint64_t contentLength,
                   [...], void *callbackData)

void S3_get_object([...], const char *key,
                   uint64_t startByte,
                   uint64_t byteCount,
                   [...], void *callbackData)
```

Another significant difference between file system and object storage interfaces concerns consistency of read/write operations. After Brewer's CAP theorem *[Bre00]* in a distributes system at most two of the following tree characteristics can be achieved:

- Consistency: by performing a read operation a process receives the most recent written data or an error.
- Availability: every read or write receives a non-error response, without guarantee that it contains the most recent written data.
- Partition tolerance: a system continues to operate in the desired way, despite of failures concerning parts of the system.

This led to a definition of an alternative semantics to characterize consistency of a systes. In contrast to the well known ACID (Atomicity, Consistency, Isolation, Durability) semantics, object storage are designed to follow the BASE semantics. BASE semantics means:

- Basically Available: a system is guaranteed to receive a response. This response can however be an error response or the requested data can be in a inconsistent or changing state.

- Soft state: the state of the system is always soft in a sense of even if there is no operations taking place, there may be changes going on due to eventual consistency.
- Eventual consistent: a system continues operating, not checking the consistency of every transaction before it moves to the next one.

File system interfaces define strongly consistent semantics. Thereby the result of any write operation done by one process is seen instantaneously by all processes operating on the same data. This requires additional locking overhead by the operating system and has additional negative impact on storage performance.

Object storage systems relax the consistency semantics for the benefit of performance. Object storage is known to be eventually consistent. This means that by making a read operation on the shared data the processes eventually receives the last updated value. In the case of object storage systems the nature of eventual consistency can lead to following inconsistent behavior:

- read after create:

  after creating an object a client immediately tries to read the same object and receives an error.

- read after delete:

  after deleting an object the same object can be still read for a portion of time.

- read after overwrite:

  after overriding an object with a new version of the same, subsequently reading the object receives the old version of it.

- list after create:

  after creating an object the response of operation for listing of all objects does not include the object just created.

- list after delete:

  after deleting an object the same object is still present in the list of all objects for a portion of time.

The exactly set of inconsistent operations varies from implementation to implementation of the object storage service *[S3I] [WLBM07] [MSA]* and has to be investigated for every single implementation.

2.3 Object Storge Interfaces and Implementations

Many different implementations providing different object storage interfaces has been developed. Technically they rely on very different storage technologies and provide a vary varying set of features. It reaches from simply implementing the API leveraging local file systems for storing data *[min]*, up to providing additional services and implementing alternative storage beckends

[Cepd]. Depending on the software used, a subset of the following properties can be supported (this list illustrates properties, which could be relevant in context of high performance computing and is not meant to be complete):

- authentication and authorization
- monitoring
- scrubbing
- encryption
- logging
- replication
- policy management
- usage tracking
- asynchronous replication over WAN
- additional interfaces like POSIX-semantics/interface and block-service *[Cepb]*, *[Cepc]*
- e.t.c.

Commonly, object storage services provide a single interface for accessing unstructured data, which resides on many storage devices under a single service point. By building a ring of storage devices, object storage services map every object to one of the devices leveraging the key of the object. By distributing objects in this way several goals are achieved:

- scalability of storage amount by eliminating file system limitations (e.g. number of files, size of file system).
- scalability of read and write operations by distributing them to a set of devices.
- potential additional scalability of read operations by replicating a single object to several storage devices on distinct servers.
- high availability ensures the operational functionality in case of failures.

Despite of the fact that every object storage system potentially provides its own set of interfaces and defines its own APIs for object and bucket operations, the intersection of this set seams to be the S3 (Amazons Simple Storage Service) interface, which is also a de-facto standard interface for operating with object storage services. Other object storage systems that implement their own interfaces usually provide an additional service which maps the S3 API calls to to the API call of the relevant interface and provide only the subset of S3 API (e.g. swift, rados).

In the following, the parts of the S3 API, which are relevant for storage operations are described. Other operations (e.g. operations dealing with ACLs, lifecycle and policy management, logging, static websites, encryption and many other) do not deal directly with data itself are usually not supported by other object storage systems and considered out of scope of this work. The description is related to the 2006-03-01 version of the API.

The S3 API is relies on HTTP/HTTPS protocol. It provides a Representational State Transfer (REST) based and Simple Object Access Protocol (SOAP) based endpoints for managing data. The API defines mainly two data structures - objects and buckets. Objects represent a portion of data accessed by the object name (a key) and can optionally have metadata attached. For separating the namespace of objects, buckets are used.

Below, basic CRUD operations on objects and buckets of the S3 API are listed. For shortening reasons the optional headers, as well as some other header which are not mandatory or common for every operation (e.g. `Authorization`, `Host` headers) are not listed. For the same reason response of the operations is skipped either.

2.3.1 Operations on buckets

- Create bucket: creates a bucket optionally setting an Access Control List (ACL) and bucket location.

```
PUT / HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com

Content-Length:  0
```

- List bucket content: list object names of the objects, contained in the bucket. Optionally additional listing parameter specifying the parameter of the returned list can be set.

```
GET / HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com
```

- Check bucket existence: determines if a bucket exists.

```
HEAD / HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com
```

- Delete bucket: deletes a bucket named in the `Host` header. For successful operation bucket has to be empty.

```
DELETE / HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com
```

2.3.2 Operations on objects

- Upload Object: performs a write operation on an object. Optionally additional header can be set in order to assign Access Control List, alternative storage class. Additionally metadata can be assigned to the object by specifying additional headers.

```
PUT /<objectname> HTTP/1.1
```

```
Host:  <bucketname>.s3.amazonaws.com

Content-Length:  <Length of uploaded content>
```

- Download object: performs a read operation on an object. Optionally some header can be set in order to control the range of read data and to provide some conditional filters.

```
GET /<objectname> HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com
```

- Delete Object: deletes an object. Optionally if object versioning is used delete all the versions of the specified object.

```
DELETE /<objectname> HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com

Content-Length:  length
```

- Get objects metadata: retrieves metadata associated with an object. Optionally some header can be set in order to provide some conditional filters.

```
HEAD /<objectname> HTTP/1.1

Host:  <bucketname>.s3.amazonaws.com
```

2.4 MPI I/O

Message Passing Interface (MPI) is an open-source standard for communication designed for inter-process communication in highly distributed parallel computing environments. The main purpose of MPI is the providing a synchronized access to shared resources in the computing cluster. MPI defines a set of communication patterns for communicating in the cluster also also allows dynamic process management.

MPI I/O is a set of functions, which allow coordinated parallel access to files residing on shared file systems. The main goal of the MPI I/O is to utilize parallelism to increase I/O bandwidth. It also provides coordinated access in order to force all processes to read/write data simultaneously or to wait for each other. The Implementations of MPI also optimizes read and write operations with the goal of efficiently servicing the requests. This happens by combining the requests of all processes and merging them.

Fig. 2.2 depicts how the MPI library and fit in the context of highly distributed computing and storage systems.

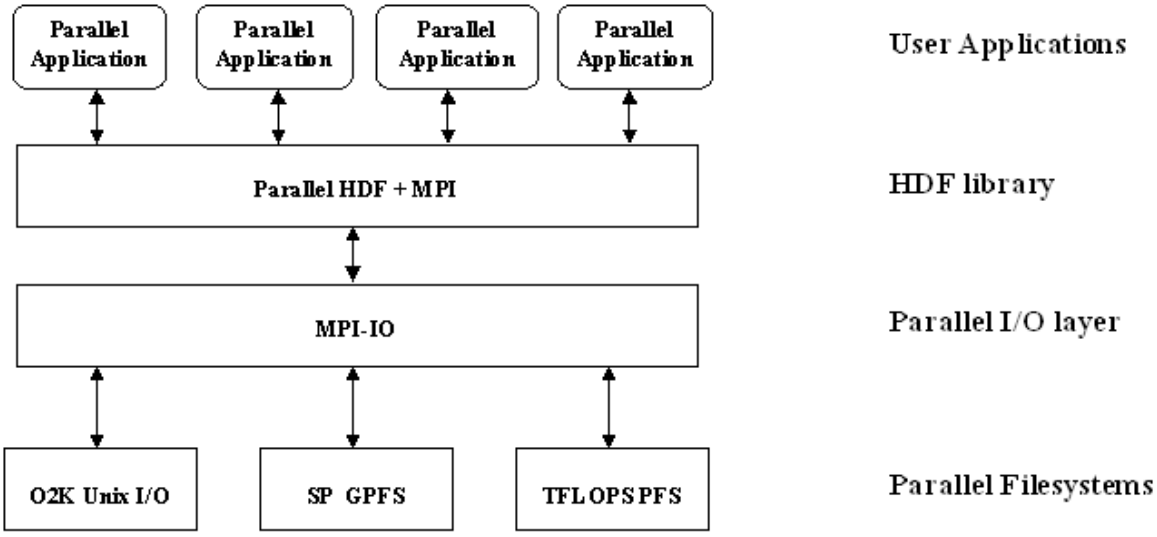

Fig. 2.2: Architecture of parallel I/O access using HDF5 and MPI.

2.5 HDF5

This section describes the abstract data model of the HDF5 library. The programming model for working with the HDF5 library is described in the next section.

The abstract data model of the HDF5 library defines the following data structures, which are used for organizing data within HDF5 file format. In many cases the tree structure is very similar to the file system structure. HDF5 files are similar to file systems, groups are analogous to the directories and datasets can be seen as files.

- **File**:

  Files represent containers for storing groups, datasets and other objects. They also provide a root group in which all other objects are stored. Additionally files store property lists which hold extra portion of information (metadata) about accessing and creating file objects.

- **Groups**:

  Groups are primary HDF5 objects. They represent containers for storing other groups and datasets and thereby provide a basis for hierarchical structure of HDF5 files. They are used to aggregate objects which logically belong together by the kind of data.

  Within a HDF5 file exist a minimum of one group - the root group with the name "/". This group represents a basis for further hierarchy. Each group can contain zero or more objects and each object must be a member of at least one group. In contrast to file system structure, which is strictly hierarchical, this allows a definition of hierarchy with circular references.

Analogous to files, groups also store property lists with additional metadata.

- **Datasets**:

  Datasets are objects where data is actually resides. As in case of files and groups additional metadata is stored next to the data itself. The metadata belonging to a dataset includes property lists with information about accessing and creating the dataset object and information allowing the interpretation of data - datatype and dataset.

  The data itself is stored blockwise without any interpretation.

- **Datatype**:

  Datatypes describe a single datapoint. They provide information about the size of each datapoint, together with the information about the layout of bits used to store the datatype. This allows the interpretation of datapoints and, if needed, also conversion of datatypes.

  HDF5 library defines a set of built-in datatypes. Additionally it allows defining own datatypes by composing the built-in types. The following list consists of datatypes which are natively supported by the HDF5 library.

  - Integer
  - Float
  - Character
  - Date and Time
  - Bitfield
  - Opaque
  - Enumeration
  - Reference
  - Array
  - Variable length
  - Compound

- **Dataspace**:

  Dataspaces represent the shape of the data stored in the related dataset. It includes information about the number of dimensions and the size of each dimension of the data, stored in the dataset.

- **Attributes**:

  Attributes are user defined key/value pairs which can be associated with a group, dataset or named datatype.

- **Property lists**:

Property lists are collection of parameters controlling the behavior of the library. Each object type has its own set of properties which can be stored in the related property list.

- **Links**:

  Links are used to interconnect the objects within the HDF5 hierarhy.

To summarize the points mentioned above Fig. 2.3 shows the hierarchical structure of HDF5 file.

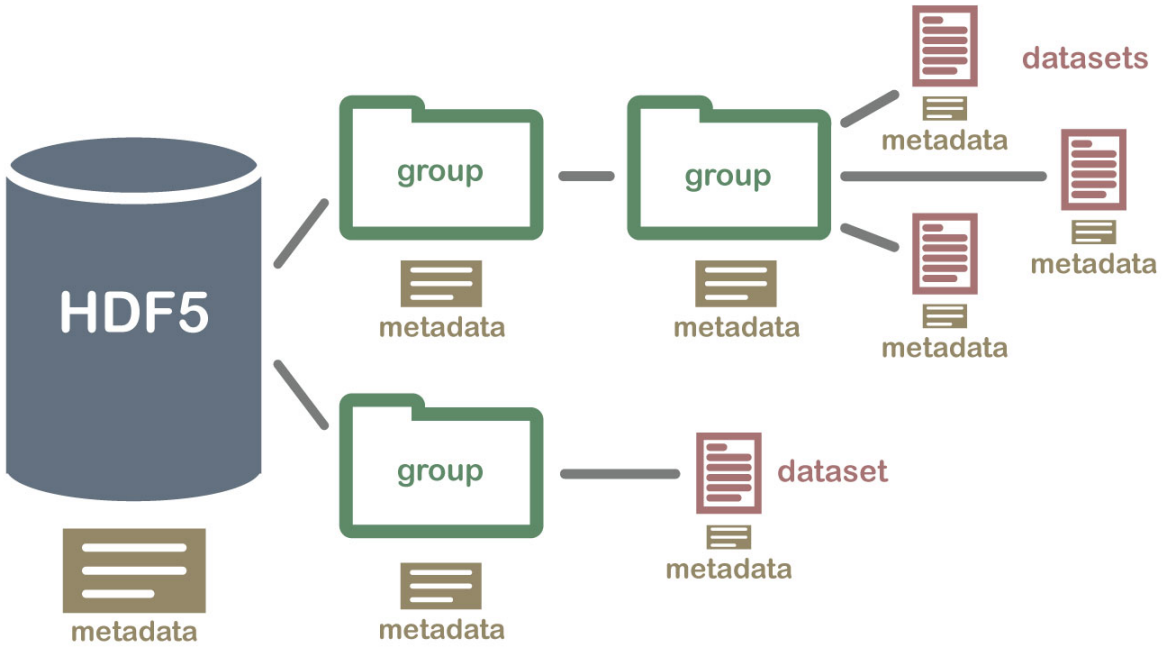

Fig. 2.3: Hierarchical structure of HDF5 file.

2.6 HDF5 VOL

Virtual Object Layer(VOL) is a new abstraction layer in HDF5 library. Residing right behind the HDF5 public API it intercepts all API routines that could potentially modify the HDF5 file and forwards those calls to the plug-in configured to used by setting the appropriate properties in the file access property list. The archtecture of HDF5 Virtual Object Layer object drivers is depicted in Fig. 2.4.

HDF5 VOL is similar to Virtual File Layer (VFL) *[HDF Group12b]* which provides an abstraction layer over file storage mechanisms, Virtual Object Layer provides an opportunity for custom handling of various components of HDF5 data model. This allows to address some of bottlenecks of the storage system as well as to provide some new opportunities for data management.

By using the HDF5 VOL one could for instance implement an alternative storage backend extra for metadata storage (due to limited performance of the same). Another desirable purposes are

stacking and mirroring of plugins. This would allow various object types in the data model to be handled by different plugins potentially reducing the complexity. One more purpose could be the storing a logging information to a remote logging server. This could be useful for later statistical analysis with the goal to optimize the backend storage of file/dataset creation or access properties. Mirroring of plugins would allow parallel writing of data in different formats.

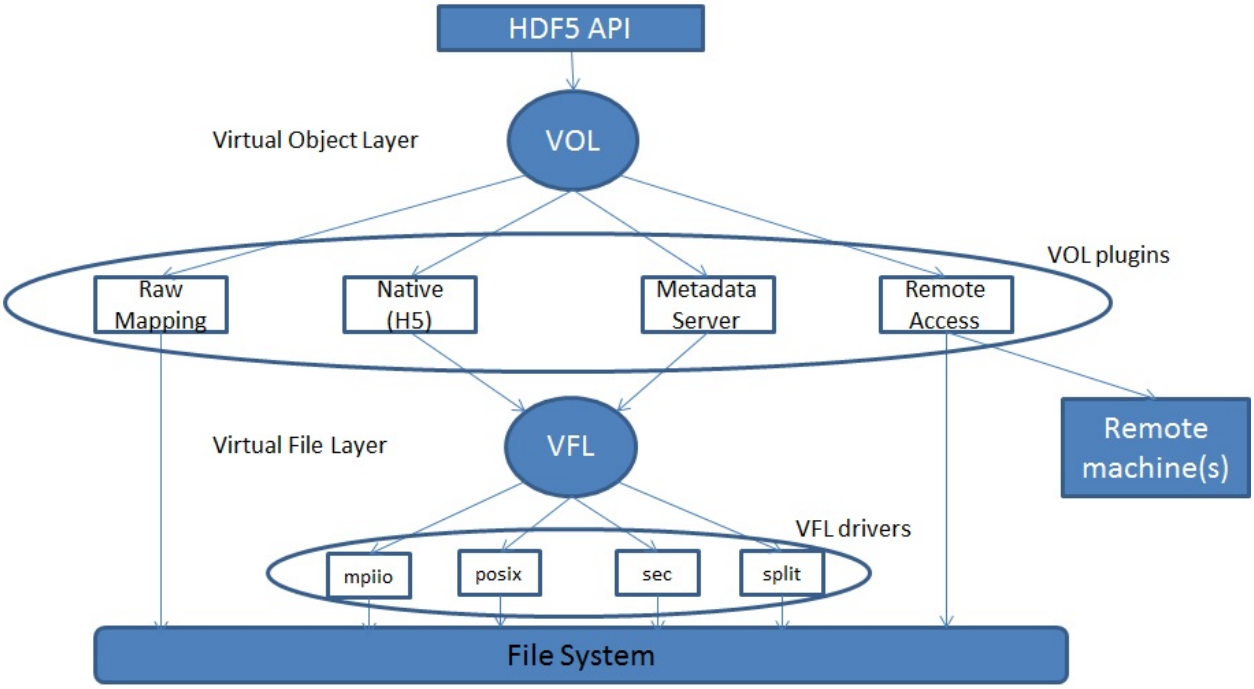

Fig. 2.4: Architecture of HDF5 Virtual Object Layer

The programming model for working with the HDF5 library defines a set of functions and programming patterns for working on various objects defined in the last section building an interface for using the HDF5 library. Basically this functions implement the CRUD operations as well as some additional functions for manipulating objects:

- create, open and close files, groups, datasets, datatype
- read and write operations to datasets
- copy operation on links
- requesting of additional parameter of files, groups and datasets
- possibility to implement driver/plugin dependent functions for files, groups, datasets, attributes and links

The additional concepts include:

- uniformly using object ids for addressing objects
- uniformly using access property lists for supplying creation, access and transfer parameters to created objects

- possibility for using asynchronous transfer mode

2.7 Summary

In this chapter the stack of technologies is for accessing data in high performance computing is introduced and every single component of the architecture is described. Additionally, the differences between the classical storage access and the one based on object storage services are stated out.

CHAPTER

THREE

PERFORMANCE EVALUATIONS OF OBJECT STORAGE SERVICES

This chapter covers performance consideration of using object storage. At first, common factors of storage performance are briefly introduced. After that, the setup of the physical infrastructure and the ceph object storage service which are used for the evaluation purposes is described. Subsequently, the evaluation results are introduced.

3.1 Performance Factors

There are plenty of factors which directly influence the I/O performance of software system using an object store. They involve hardware and software components of systems, but also consider possible implications of of logic, implemented by an end user application. In the following various parts of object storage architecture together with performance influencing elements are listed, grouped in categories by system component. Each element of this list represents a possible optimisation vector and can be tuned within the scope of possibilities. This list should generally give an overview of possible influencing factors and should not be considered as complete.

3.1.1 Server Performance Considerations

Aspects mentioned in this section concern all storage systems and are valid not exclusively for object storage systems. This parameters concern mainly the hardware and the configuration of storage systems and can be adopted with some effort by the operational side.

- CPU usage and capabilities

  The time needed for processing a computing operation is determined by the CPU capabilities and is essential for performance of storage systems.

- Memory usage

  The amount of free memory determines the caching capabilities of system. Since the access to the main memory is much more efficient, the performance acceleration can be enormous.

- I/O subsystem
  - Concurrency

    The amount of concurrent operation on storage devices have a significant influence on the common performance. Depending on access pattern and the I/O subsystem some system are capable of optimizing the access by reordering the operations.

  - Access time

    Access time is the delay which is introduced by several components of the storage systems. Storage devices with non random access to the data (e.g. spinning disks) require some portion of time to move the access arm to the position where data is stored/has to be written. Network storage system potentially add additional delay by establishing a connection and requiring some sort of authentication and encryption.

  - Throughput

    After establishing the connection or positioning the access arm to the right position, the performance of reading or writing operations is known as throughput. This performance can also as well depend on some factors (e.g. file system alignment, data fragmentation and other).

- I/O strategy
  - Caching

    Holding the most frequently accessed data on performant storage or even in memory is a common practice in computing environment. To predict the subsequent data access physical and temporal locality of data is used. Bu using cache mainly two goals achieved. By using a read cache no operations have to be delegated to a slower medium. Using a write cache helps to keep the underlying storage constantly busy. This is especially useful for workloads which have a burst character. In both cases the workload on the underlying storage is reduced.

  - Replication

    Creating local (RAID) or distributed copies of data also contributes to performance improvement due to enhancement of throughput. However this strategy should only be used in environment with mostly reading workloads, thus keeping the replica consistent requires additional overhead.

  - Aggregation and reordering of requests

    Additional improvement of I/O performance can be achieved by this strategies. In case of aggregation, several operations are combined to a bigger one if it is expected to increase the performance. Therefore the operation on the storage devices can be reordered with the goal to create a most contiguous order of storage accesses.

- Architecture
  - I/O control

By I/O control a piece of hardware is meant, which takes over a particular task relieving the CPU. (e.g. RAID-Controller, DMA)

- Bus system

  Bus system defines the interconnection between components within a single machine. The specifications of bus systems defines also the maximal throughput of communication between this components(e.g. SATA vs. NVMe, QPI vs. FSB).

3.1.2 Interconection Performance Considerations

Aspects mentioned in this section concern all network storage systems and are valid not exclusively for object storage systems. This parameters concern both hardware as well as software components of the communicating systems. Some aspects can be adopted with some effort by the operational side, other require additional effort from the client side.

- hardware
  - Topology

    Topology is defined as the logical and physical structure of network components including interconnections. Some of the topologies allow directly communication between the components, not sharing any common resources (e.g. fully mashed networks). Other network topologies share common resources which introduces additional delay in processing network I/O.

  - Latency

    In the network file system latency has a significant influence for operations on small objects like metadata. Operating on big data portions becomes negligible.

  - Buffer size

    Networking hardware like switches usually have an amount of memory for temporarily caching I/O operations. This allows queueing and reordering of network packets/frames. In big networks with heigh network load, small amount of memory used for caching could also have a negative influence of the performance.

  - Bandwidth

    Bandwidth refers to the data transfer rate supported by network interconnection and represents the maximum capacity of it.

- software
  - Protocol

    Protocols define a common communication pattern between network components. They define the establishment of the connection and define additional guarantees for data transfer. This factor can also influence the common performance of the storage systems.

- Frame size

  Setting the size of network transmitting unit to the most possible value could additionally contribute to the performance of large read and write operations due to reducing the ratio of network header to payload.

3.1.3 Client Performance Considerations

Aspects mentioned in this section concern all systems. They are controlled by only by user application, performing operations on the storage systems. From the operational side there is no possibility to influence this parameters.

- Access pattern
  - Type of operation

    Read and write operation stress storage systems in a different way. Usually write operations tend to be more performance intensive.

  - Size of requested data block

    High I/O throughput is reached especially when large operations are performed. This minimizes a ratio of the latency to the overall overhead.

  - Contiguity of access

    Using storage devices with no random access features, contiguity of access is very important for overall storage performance. The best performance is reached when several portions of requested data are physically located closely together on the persistent storage, building a contiguous access. In case when several parts of a data are distributed along a storage device, in addition to the extra overhead for seeking every piece of data, it also has to be recomposed together. This requires a in-memory copying.

  - Predictability of access

    Predictability describes a mass of similarities between the operations of an application over time. The more predictive a pattern is, the more comprehensive I/O optimizations can be exposed.

3.2 Setup of Evaluation Environment

The object storage service of choice to evaluate is Ceph. Ceph is an open-source object storage system which can build up to exabyte-scale storage system. It is completely distributed and avoiding single points of failure. Providing a set of functionalities common for every distributed object storage system Ceph implements also two unique features providing file and block storage interfaces based on underlying object storage Fig. 3.1 *[Cepa]*.

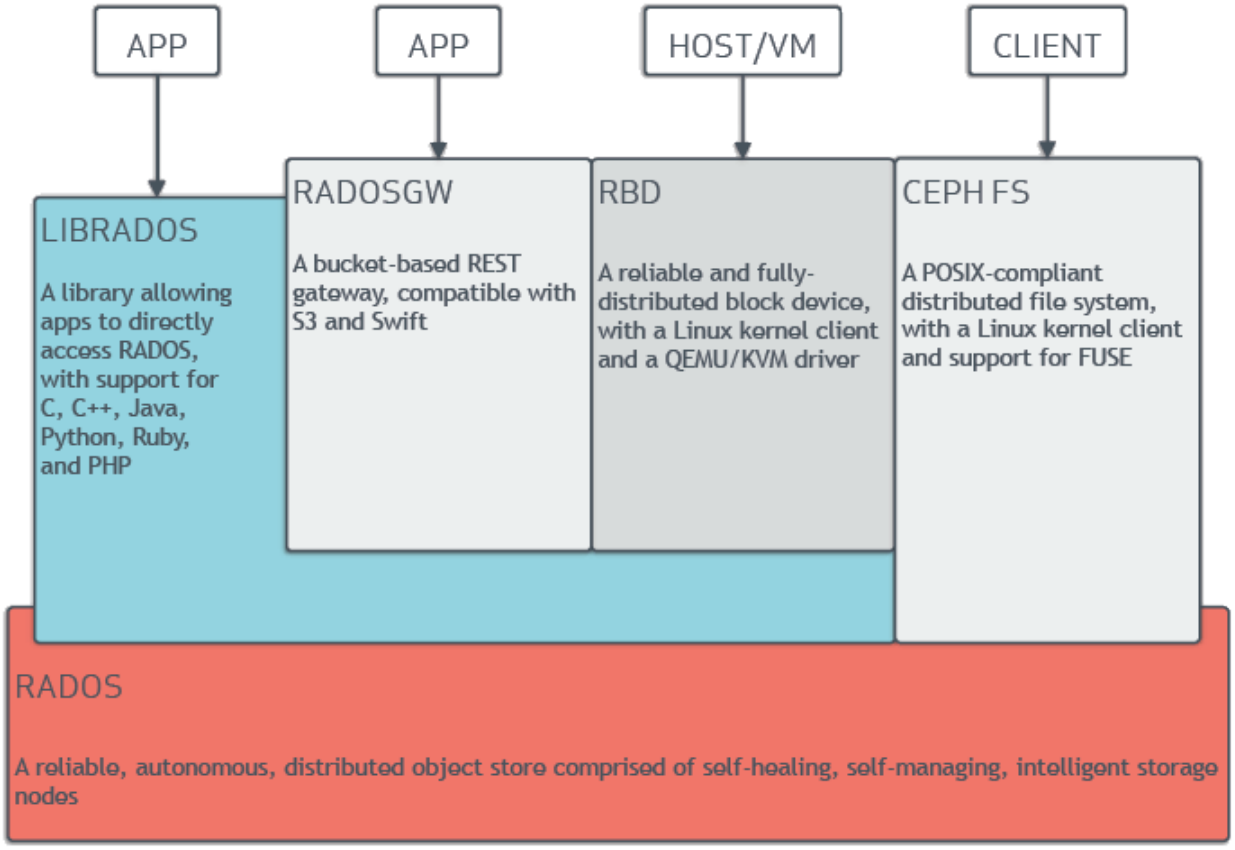

Fig. 3.1: Interfaces provided by Ceph object storage cluster.

Ceph is built upon CRUSH (Controlled Replication Under Scalable Hashing) algorithm *[WBMM06]* - a hashing algorithm which efficiently maps objects to storage devices in a cluster. CRUSH also has a notion of hierarchical structure - "a CRUSH map" of a storage cluster and object replicas. This allows distributing objects and its replicas to different storage devices which have no shared resources (e.g. I/O bandwidth, network bandwidth). Thus by creating copies of each object and placing them on different Failure Domains (disks, servers, racks, cabinets) high availability is reached.

By creating an object on Ceph storage cluster a client computes a placement group id using objects key. After that, using information from clusters monitor about the state of the cluster and the CRUSH map, the client determines the exactly device to read to/write from and communicates directly to this device.

To provide object storage functionality the following components of Ceph storage architecture are involved:

- Monitor: monitoring processes keep track about the cluster status, monitor resource usage, negotiate consensus about the master cluster map between multiple monitor processes, receive heartbeats from other components of the system.
- OSD: each storage device is represented by a Object Storage Daemon (OSD) building a logical disk. OSD processes make storage devices accessible over network through RADOS protokoll.
- RadosGW: Rados Gateway processes provide S3 and Swift API's. Working as proxies this processes mediate between this API's and RADOS protokoll.

The evaluation environment consist of six Sun Fire X2200 M2 (TODO: link to specs) Servers with the following hardware configuration:

- 2 x Quad-Core AMD Opteron 2376
- 32 GB Memory
- Broadcom BCM5715 Gigabit Ethernet
- Western Digital WD1003FBYX, 1 TB, 7200 rpm

All server are interconnected through ProCurve J4904A 1Gb/s switch building a storage network. As operating system Ubuntu 16.04 LTS with kernel 4.13.0 is used.

Three of the servers acts as clients performing read and write operations, the other three services build a storage cluster. In order to provide no bottleneck all the three services required for operating the object storage service are deployed on every node. Each client is connected to the different server.

Unfortunately, no benchmarking tool could be found to support all of the object storage interfaces supported by Ceph. However, the "Yahoo! Cloud Serving Benchmark" (YCSB) is capable of benchmarking of Rados and S3 interfaces and provides a simple and clean programming model to allow the implementation of additional benchmark of Swift interface using the `openstack4j` library.

YCSB is a framework for benchmarking of database systems. It includes an extensible workload generator and a set of basic workload scenarios to be executed by the generator. Currently is supports over 30 different database interfaces and is also capable of benchmarking of some other interfaces like generic REST-based interfaces. Amongst other interfaces it provides support for Rados- and S3- APIs.

3.3 Performance of Object Storage

In this section performance capabilities of the evaluation environment are investigated.

The main purpose of the this benchmarking is to investigate, how an object storage system behaves under a heavy parallel load. Therefore a set of parameters is defined under which an Object storage API is put under stress testing. The following parameter and corresponding values are chosen:

- Type of performed operation:

  As stated above different types of operations generate a very different load on storage systems and both types of operations, read and write are included in the parameter list.

- Concurrency of storage access:

  In context of high performance computing storage systems are exposed a continuous load from thousands of clients. A storage system deployed under in such environment has to continue working and provide a respectable performance to the clients. Following values for the number of parallel processes is chosen: 3, 30, 60 and 90.

- Object size:

  The relevance of sizes of accessed data is also described above. In order to cover the whole bandwidth of object sizes the logarithmic scale of this axis is chosen. The exact values for this parameter are the following: 1Mb, 10Mb, 50Mb and 100Mb.

The main goal is the investigation of parameter space in order to find out the optimal parameter in dependency of parameters described above and to develop a model for prediction of performance with the goal of minimizing latency and maximizing throughput.

The Benchmarking results for the S3 API are shown below. Benchmarking of Swift and Rados interfaces show very similar results and therefore skipped. The representation outliers is skipped for the viewability reasons.

3.4 Analysis of Benchmarking Results

Reviewing of the benchmarking results lead to the following observations:

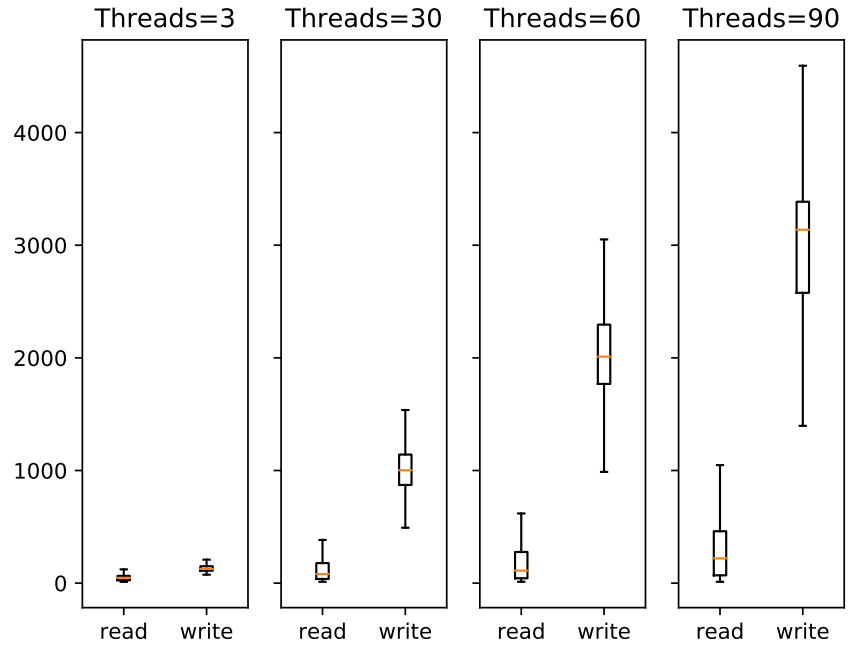

Fig. 3.2: Latency of object storage systems dependent on number of parallel storage operations with object size 1Mb

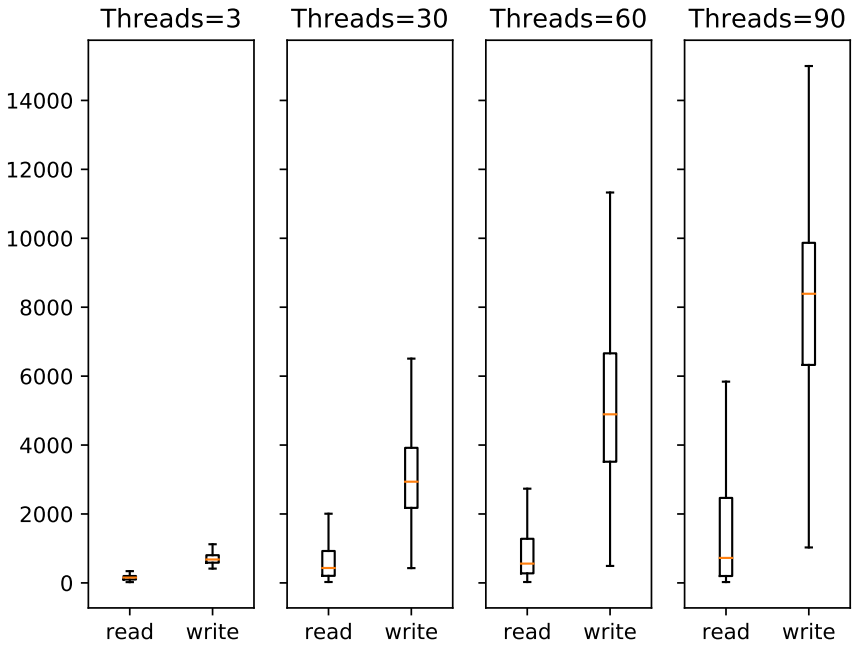

Fig. 3.3: Latency of object storage systems dependent on number of parallel storage operations with object size 10Mb

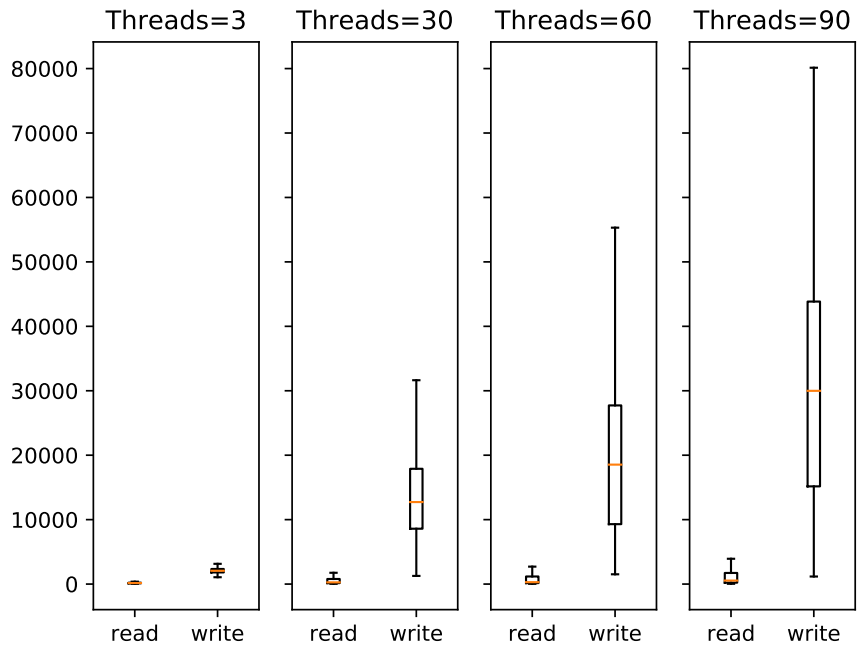

Fig. 3.4: Latency of object storage systems dependent on number of parallel storage operations with object size 50Mb

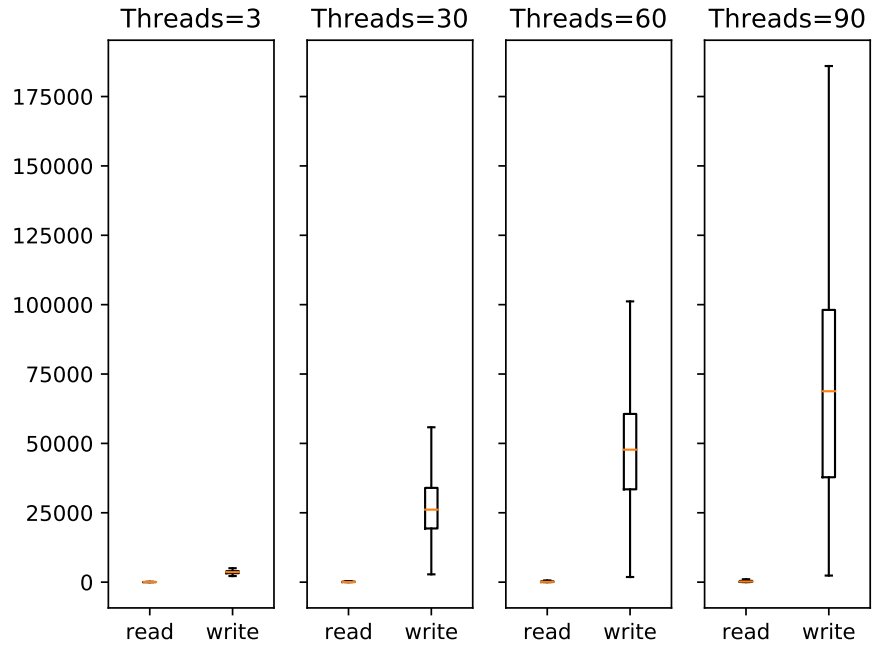

Fig. 3.5: Latency of object storage systems dependent on number of parallel storage operations with object size 100Mb

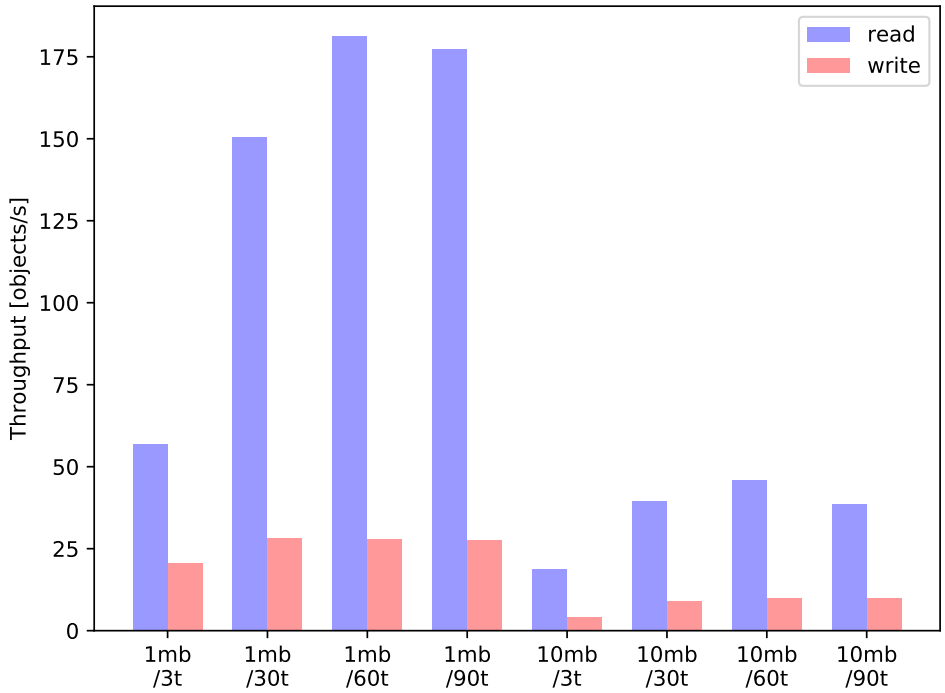

Fig. 3.6: Throughput of object storage systems dependent on number of threads. Object sizes 1MB and 10MB.

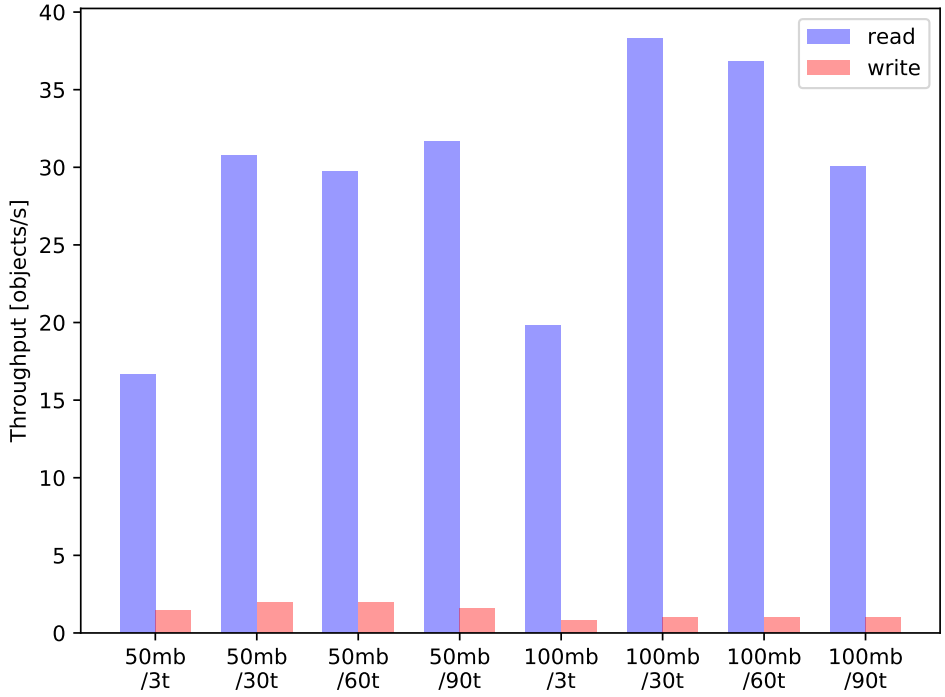

Fig. 3.7: Throughput of object storage systems dependent on number of threads. Object sizes 50MB and 100MB.

3.4.1 Throughput

- With increasing number of parallel processes the throughput of read operation is increasing. For large enough object sizes a 1Gb/s ethernet network can get fully saturated.
- With increasing number of parallel processes the throughput of read operation rapidly increasing up to a value of about 50 threads. From this point the throughput stay relatively constant.
- The course of the functions for read and write operations is very similar and seen to correlate.
- The ratio of throughput of read operation to the throughput of write operation seems to behave logarithmic dependent of the number of threads and of size of objects.

3.4.2 Latency

- The slope of the function of latency of writing operation is much more steeper then the one of the reading function in dependency of threads..
- With increasing number of threads the scattering becomes bigger.
- With increasing number of threads the average latency grows. This seems to behave near liner in both cases, for writing and reading and dependent on both - the number of threads and the size of objects.

3.5 Summary

In this chapter common performance considerations in distributed systems are covered. After that, the setup of the physical infrastructure and the ceph object storage service which are used for the evaluation purposes is described.

CHAPTER

FOUR

DESIGN

This chapter documents various design decisions that have to be made in order to implement the S3 VOL plugin.

4.1 Mapping HDF5 Objects to Object Store

There are differences between how data is organized in HDF5 file format which do not allow it do be mapped to object storage directly. Basically HDF5 file format defines two datatypes to represent hierarchical relation: datasets and groups. Datasets represent collections of information of the same type usually organized in multidimensional space. Groups are used to compose similar datasets. To access a dataset in the hierarchical structure of a HDF5 file one have to define a relative of absolute path under which the dataset is located (similar to accessing a file on Linux/Unix file system).

Object storage services use a different way to reference objects. Due to the fact, that object storage does not implement hierarchical relations and object access is done by the key of corresponding object, one has to adopt the addressing of HDF5 objects when object storage services are used as backend.

Additionally files, groups and datasets have some extra portion of metadata like property lists, datatype and dataspace, which have to be stored extra. Object storage services provide a possibility to store small amount of data as objects metadata. This could be used to store metadata for groups and datasets, however the size of this data is quite small (2kB in S3 implementation *[s3d]*) and varying dependent of actual implementations and configuration of object storage service. In addition, S3 interface does not support updating and deleting of metadata. The whole set of key/value pair building the metadata has to be defined by the time of creation of the object (subsequently adding a metadata to an object requires copying it on the server side). Since it would be inefficient and in order to keep the implementations suitable for alternative implementations of object storage service, the decision was made not to rely storing HDF5 metadata as objects metadata.

In order to ensure backwards compatibility the HDF5 has to deal with the changed layout of storing information and has to support the same addressing manner as the HDF5 library. However object storage services have very little restrictions on naming of object keys. This allows one to use the

full path to the object as a part of the key. Additionally metadata for groups and datasets is stored as extra objects prefixed with the key of the corresponding object.

In the following two example layouts of the same structure stored in object store Listing 4.2 and as plane HDF5 file Listing 4.1 are shown. The structure of HDF5 is shown as the output of `h5dump` command. The listing of resulting objects stored in object store is shown as the output of `s3 list` command. File is mapped to S3 buckets, groups are mapped as aprfix of the object key and datasets are decomposed and stored as objects.

Listing 4.1: Example of hierarchcal structure defined by native HDF5 file format as returned by `h5dump` command.

```
$ h5dump dset.h5
  HDF5 "dset.h5" {
    GROUP "/" {
      GROUP "mygroup" {
        DATASET "mydataset" {
          DATATYPE  H5T_STD_I32BE
          DATASPACE  SIMPLE { ( 4, 6 ) / ( 4, 6 ) }
          DATA {
            [...]
          }
        }
      }
    }
}
```

Listing 4.2: Example of hierarchcal structure of HDF5 file format mapped to Object storage as returned by `s3 list` command.

```
$ s3 -u list dset
                        Key                    Last Modified       Size
-------------------------------------  --------------------  -----
fapl                                   2018-01-07T10:48:38Z   1516
fcpl                                   2018-01-07T10:48:38Z    412
gapl                                   2018-01-07T10:48:38Z    113
gcpl                                   2018-01-07T10:48:38Z    113
mygroup/gapl                           2018-01-07T10:48:38Z    117
mygroup/gcpl                           2018-01-07T10:48:38Z    113
mygroup/mydataset_0000000000           2018-01-07T10:48:38Z     48
mygroup/mydataset_0000000001           2018-01-07T10:48:38Z     48
mygroup/mydataset_dapl                 2018-01-07T10:48:38Z    210
mygroup/mydataset_dcpl                 2018-01-07T10:48:38Z    132
mygroup/mydataset_space                2018-01-07T10:48:38Z     63
mygroup/mydataset_type                 2018-01-07T10:48:38Z     14
```

One additional design consideration concerns defining a set of variables which handle are read by the plugin and interpreted as information required for authentication against the object storage system and specifying the location with the endpoint of the service.

4.2 Mapping Memory Space to Object Store

The default behavior of the HDF5 library is to stores datasets in contiguous layout. As depicted in Fig. 4.1 this is done by mapping memory space directly to file space.

As already described in the last chapter, for efficient access the space of every dataset has to be partitioned into smaller object. HDF5 library already supports splitting a dataset into a partitions by chunking mechanism. This is done by supplying additional parameters to dataset creation property list. The resulting layout is shown in Fig. 4.2.

Chunking is used by by HDF5 library in many contexts. Chunking allow enabling compression and other filters on datasets, as well as creating extendible or unlimited dimension datasets. A built in caching mechanism based on chunking also contributes to performance improvement. Therefore using chunking for partitioning data space would bring additional features as well as provide a well known interface to the end user. With a chunked storage layout the data is stored in equal-sized blocks or chunks of a pre-defined size within the same file. The HDF5 library always writes and reads the entire chunk.

As one can see, the behavior of chunks defined by the HDF5 library have many things in common with the behavior of objects. Reusing of this functionalities would be highly desired for implementing of the S3 plugin based on Virtual Object Layer. The investigation of HDF5 library for

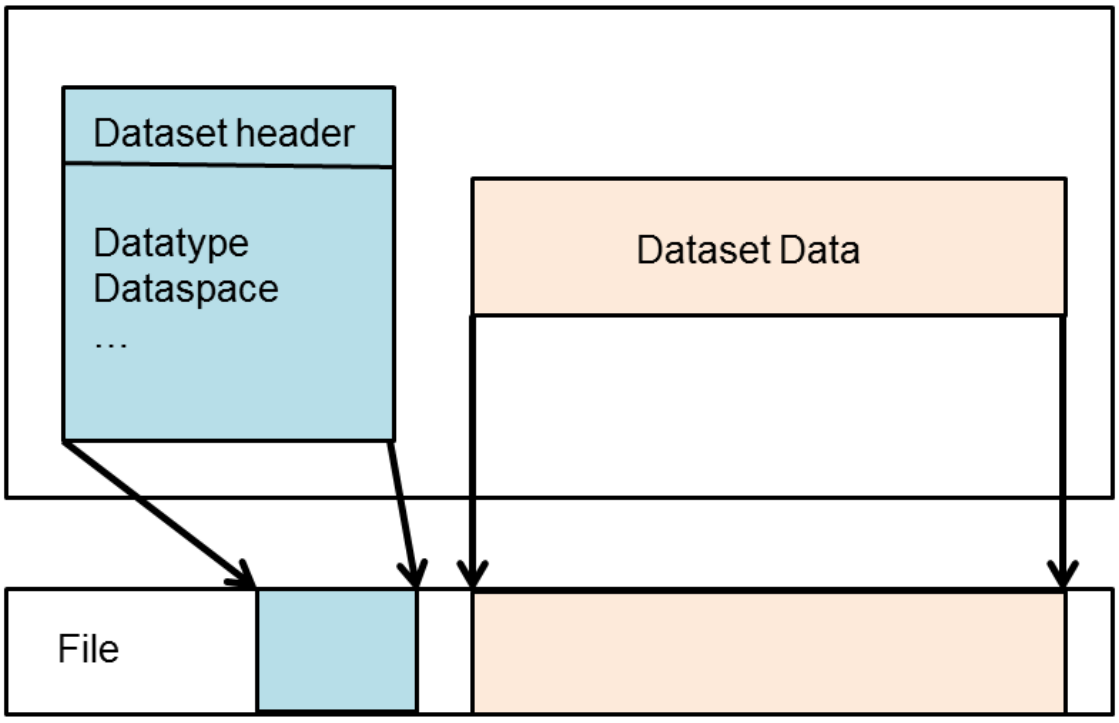

Fig. 4.1: Contiguous storage layout of HDF5 dataset

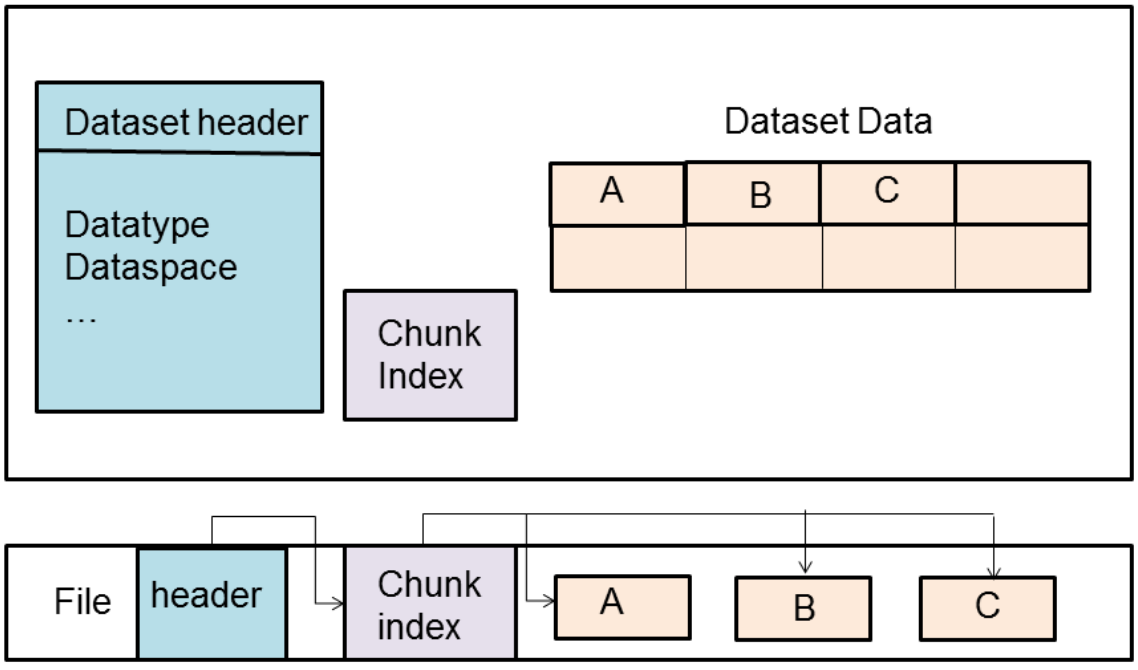

Fig. 4.2: Chunked layout layout of HDF5 dataset

reusing the same functionalities has shown that they are distributed along many files and object and are tightly bound to the internals of the HDF5 library. This makes it hardly possible to reuse the same codebase. Therefore some set of functions providing the same functionalities have to be implemented.

As shown in Fig. 4.2 native chunking mechanism uses chunk index which keeps track of the chunks associated with a dataset. Tracking chunks in the same way as it is done by the native HDF5 implementation in a highly distributed environment using a common resource would be inefficient. Therefore an other solution has to be found to map partitions of datasapce to object storage. The proposed solution for this purpose is a hash function, which computes a suffix of the object key in dependency of a region of dataset application reads from or writes to.

Additionaly, in order to support partial I/O operation a set of functions has to be implemented to support reading and writing to specified regions of a dataset, as defined by using the `H5Sselect_hyperslab` function.

4.3 Summary

In this chapter various decisions concerning the design of the S3 VOL plugin is described. A mapping of files, groups and datasets is introduced and a scheme for addressing of HDF5 objects is offered. Subsequently commonalities as well as differences in the behavior between HDF5 chunking and Objects are stated out. After it, the problems of addressing portions of datasets and selecting a single regions are discussed.

CHAPTER

FIVE

IMPLEMENTATION

This chapter provides an overview of the implementation of HDF5 VOL plugin for storing dataset in object storage system.

5.1 The Main Components

In order to implement the plugin for storing the content of datasets within a set of objects residing on distributed object storage system several components of data and programming model as desired by the HDF5 Virtual Object Layer have to be implemented Listing 5.1:

Listing 5.1: H5VL_class_t holds all implemented function for implementing the S3 plugin based on HDF5 Virtual Object Layer

```
static const H5VL_class_t H5VL_s3_vol = {
      0,                    // Class version #
      S3PLUGINID,           // value to identify plugin
      "s3vol",              // Plugin
      *H5VL_s3_init,        // init
      *H5VL_s3_term,        // terminate
      sizeof(S3_fapl_t),
      *S3_fapl_copy,
      *S3_fapl_free

    {NULL},                     // attribute class callbacks
      {                             // dataset class callbacks
          *S3_dataset_create,
          *S3_dataset_open,
          *S3_dataset_read,
          *S3_dataset_write,
          *S3_dataset_get,
          NULL,                      //*S3_dataset_specific,
          NULL,                      //*S3_dataset_optional,
```

(continues on next page)

(continued from previous page)

```
            *S3_dataset_close
        },
        {NULL}                                  //datatype
        {                                     // file class callbacks
            *S3_file_create,
            *S3_file_open,
            *S3_file_get,
            *S3_file_specific,
            NULL,                              //*S3_file_optional,
            *S3_file_close
        },
        {                                     // group class callbacks
            *S3_group_create,
            *S3_group_open,
            *S3_group_get,
            NULL,                             //*S3_group_specific,
            NULL,                             //*S3_group_optional,
            *S3_group_close
        },
        {NULL},                              // link class callbacks
        {NULL},                              // object class callbacks

        {NULL},
        NULL
    };
```

- Callbacks:

  A set of callback defied by the HDF5 virtual object layer have to be implemented. Technically, they capture all the API calls, which can potentially change the content of HDF5 and redirect them to the appropriate function.

- Datatypes:

  Datatypes for file, groups and datataset, which are required by the programming model, have to be adopted to fit the needs of the new plugin.

- S3 function:

  A set of function that actually communicate with the object storage service and process the I/O related operation have to be implemented.

5.2 Authentication

In order to perform requests to the object storage service using S3 interface every process has to have valid credentials for accessing the service. Additionally some auxiliary information have

to be provided in order to communicate the general conditions of the connection to the Object storage service. The usual way HDF5 library is doing this is by supplying this parameters in form of property lists. However, this parameters are required for the first connection, during the file opening procedure only and thereby holding them within the file creation property list makes no sense.

Specifying the connection parameter to the object storage service is done by setting environment variables to appropriate values. The whole set of environment variables is described below:

- S3_HOSTNAME:

  Provides the name of the host, where the object storage service is located.

- S3_BUCKETNAME:

  Provides the name of the bucket to operate on.

- S3_PROT:

  The default behavior of the libs3 library is to use ssl encryption to connect to the service to perform operations of bucket and objects. By setting this variable to the value of `http` no encryption is used.

- S3_URISTYLE:

  Amazons simple storage provides two ways for specifying the name of the bucket: an path style and vhost style. In case of path style specification the bucket name is served as suffix of the host name where the object storage resides segregated by a separator "/". (e.g. `s3.amazonaws.com/mybucket`). In case of vhost style the name of the bucket is specified as a prefix of the host name of service URI segregated by the separator ".". (e.g. `mybucket.s3.amazonaws.com`). The behavior described at least is the default. Setting this parameter to the value of `path` toggles this behavior.

- S3_ACCESSKEY and S3_SECRETKEY:

  This two variables actually provide the S3 credentials.

This variables are read during the initialization phase of the plugin by the function `mkContext()`, which returns a struct with various connection parameters of the object storage service connection. This struct is used by all the S3 related functions.

5.3 Operations on Bucket and Objects

Despite of the fact that the libs3 library already implements interfaces for operating on Bucket and Objects they require a specification of extra callback functions which manually handle responses of every operation (e.g. `GET`, `PUT`) on the object service and react when an operation is complete Listing 5.2.

Listing 5.2: The function `S3_put_object` of the libs3 library defines extra callbacks for custom handling of responce properties of object storage services.

```
void S3_put_object(const S3BucketContext *bucketContext,
                   const char *key,
                   uint64_t contentLength,
                   const S3PutProperties *putProperties,
                   S3RequestContext *requestContext,
                   const S3PutObjectHandler *handler,
                   void *callbackData)
```

The implemented function wraps this calls in order to provide a generic handling of responses and make a using of them more user friendly. In addition to some error handling callbacks and some other helper functions following function are implemented for this purpose:

On buckets:

- listBucket

  Lists the content of the content of the bucket specified by the arguments.

- createBucket

  Creates a bucket specified by the arguments.

- testBucket

  Tests the existence of the bucket specified by the arguments.

On Objects:

- getObject

  Downloads an object specified by a key.

- putObject

  Uploads an object to a object storage service.

5.4 Operations on Files, Groups and Datasets

For operations on files, groups and datasets the `H5VL_class_t` struct defines mainly create, open, get and close operations. The design and the implementation implementation of this functions is very similar and can be summarized. The additional write and read operations are described below.

- create operations: (S3_file_create, S3_group_create, S3_dataset_create)

By creating of a file, group or a dataset the corresponding corresponding datatypes are initialized with the parameters of the function arguments. Policy lists are stored within the storage service. In the case of dataset in addition to property lists, space of the dataset and type of the data are stored.

As explained in the last chapter the prefixes of the keys of the objects are computed based on the path to the object in the hierarchy. The suffix is determines by the type of stored data.

- open operations: (S3_file_open, S3_group_open, S3_dataset_open)

  Opening operations read the objects (property lists, space, type) stored within a object storage system. A corresponding initialized datatype is returned.

- get operations: (S3_file_get, S3_group_get, S3_dataset_get)

  A get operation returns one of the parameters of the corresponding datatype (e.g. space or type in case of datatype, or property lists). Which parameter have to be returned is defined by the arguments of the functions.

- close operations: (S3_file_close, S3_group_close, S3_dataset_close)

  For closing of a file, a group or a dataset the memory occupied by the corresponding datatypes is freed.

5.5 Partitioning a dataset

As briefly mentioned in the last chapter, partitioning of the space of dataset as required for efficient storing of data in object storage services and defining a HDF5 native chunking have many things in common. Therefore defining the partitioning parameters in same way as it is done in case of specifying chunking parameters would be obvious.

Specifying the size of chunks for datasets is done by using the function `H5Pset_chunk` as it is done in the HD5 native API Listing 5.3. This function sets the chunking related parameters of dataset creation property list. This parameters are used by the functions which actually read and write data to datasets.

Listing 5.3: Specifying of partition space of S3 objects.

```
1 herr_t H5Pset_chunk(hid_t plist, int ndims, const hsize_t * dim )
```

The parameters used by this function are:

- plist: id of the creation property list of the dataset.
- ndims: number of dimensions of each chunk. This number should match the number of dimensions of the dataset.
- dim: array which holds the number of datapoints in each dimension.

5.6 Reading and Writing Datasets

Reading and writing of data to datasets are done by selecting a region (a hyperslab) of dataspace of the dataset, where the operation has to take place. Selecting this hyperslab is done by calling the HDF5 function `H5Sselect_hyperslab`, which returns an id of the dataspace if the selected hyperslab. Using this id HDF5 library performs read or write operations.

In case of using object storage services as storage backends the selected hyperslab has to be further splitted according to the grid, defined by the chunking properties. Further, every part of the hyperslab has to be mapped to the name of the object, where this part has to be stored.

Additionally, dependent of the boundaries of the selected hyperslab, an object, to which this part of hyperslab is mapped, has to be read or written either as a whole object, ore partially. In case of writing a part of the object, before writing, the corresponding object has to be read.

5.7 Computation of Object Suffix

The following function computes the suffix of the S3 object, where the datapoint `point` should be located.

Listing 5.4: The getSuffix implements the computing of the suffix of S3 objects for every datapoint in dataset space.

```
1  int getSuffix(hsize_t ddims[], hsize_t cdims[],
2                hsize_t point[], int ndims){
3
4      int suff = 0;
5      for(int i = 0; i < ndims; i++){
6          int rest = point[i]/cdims[i];
7          suff = suff + rest * (i +1);
8      }
9
10     return suff;
11 }
```

The parameter of the function are listed below:

- ddims: a list of dimension sizes of dataset
- cdims: a list of dimension sizes of chunks
- point: coordinates of the datapoint
- ndims: number of dimensions of dataset

CHAPTER

SIX

EVALUATION

6.1 Benchmarking the implemented plugin

For the Benchmarking of the implemented plugin "IOR Interleaved-Or-Random" *[ior]* is used. This tool is designed for performance evaluation of parallel file systems. Using the implemented plugin it is also capable to benchmark the I/O operations on the test environment.

In order to make the results comparable the parameter of the benchmark have to be chosen carefully to match the parameter used by the YCSB benchmark.

After the some changes on the IOR benchmark related for using of the implemented plugin, the same evaluation platform is used as described in chapter 3.

However, IOR has no notion of objects, so the benchmarking results have to be scaled in order to be compared with the results of the YCSB benchmark.

In essence, the shape of Latency graphs remain the same. The scaling however has changes.

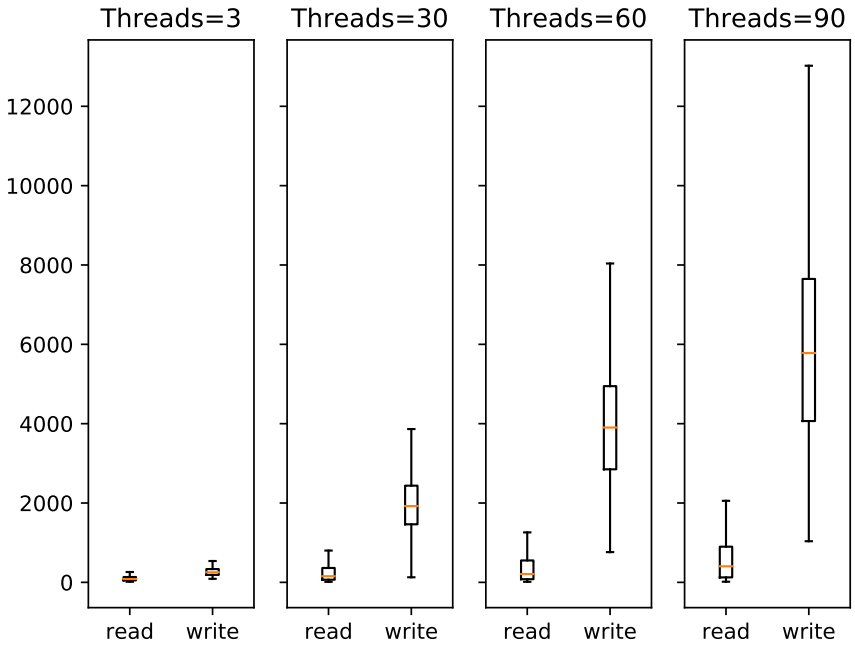

Fig. 6.1: Latency of object storage system using implemented plugin dependent on number of parallel storage operations with object size 1Mb

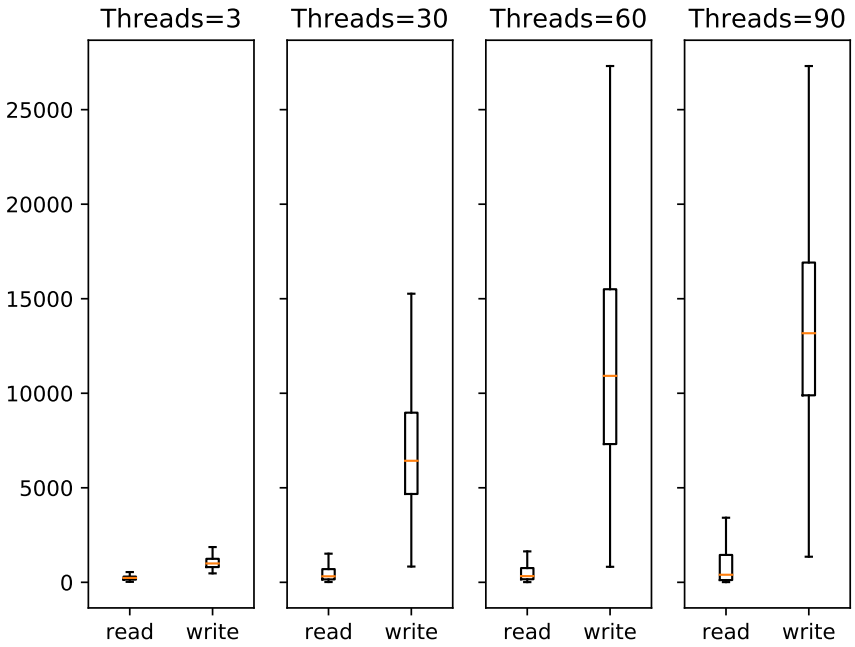

Fig. 6.2: Latency of object storage system using implemented plugin dependent on number of parallel storage operations with object size 10Mb

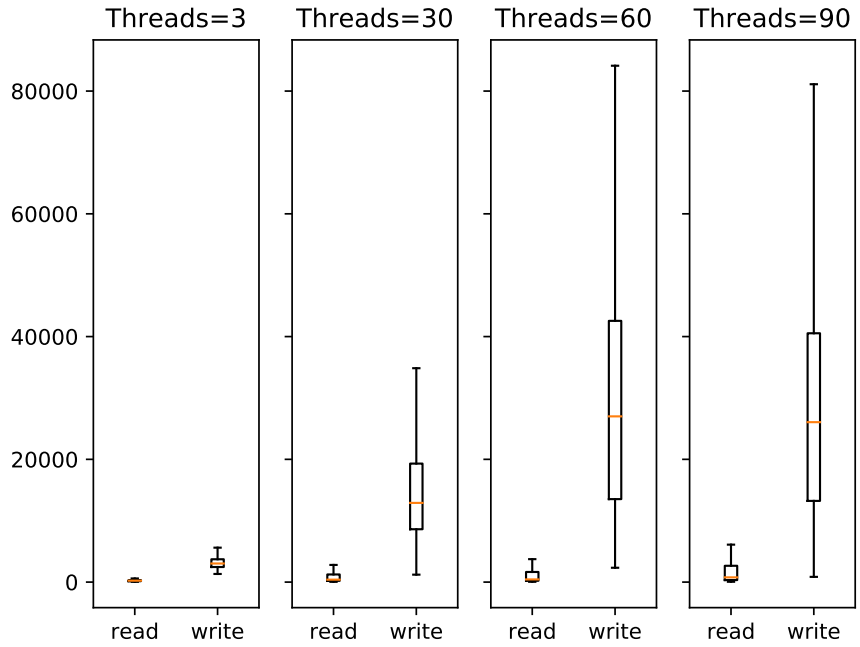

Fig. 6.3: Latency of object storage system using implemented plugin dependent on number of parallel storage operations with object size 50Mb

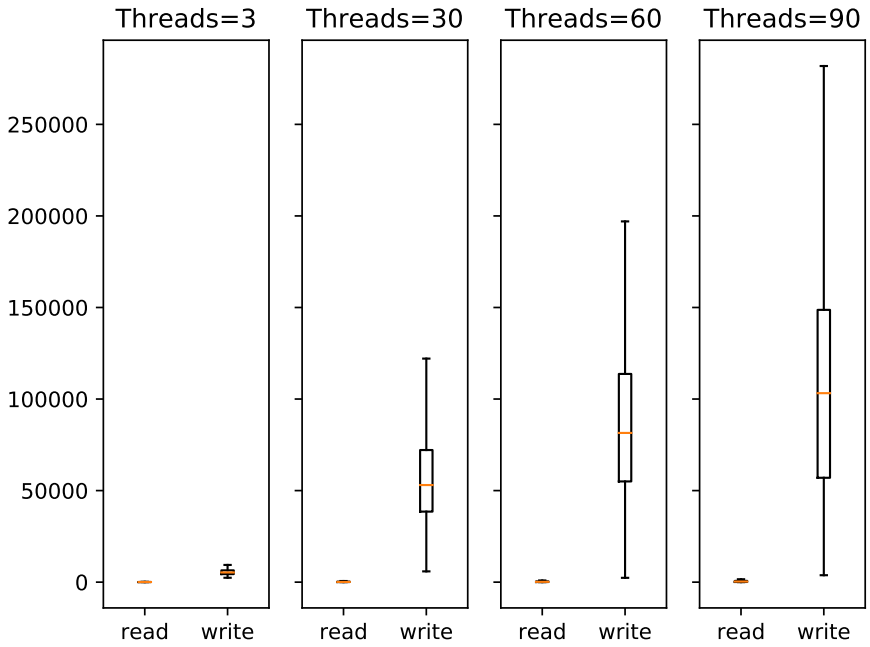

Fig. 6.4: Latency of object storage system using implemented plugin dependent on number of parallel storage operations with object size 100Mb

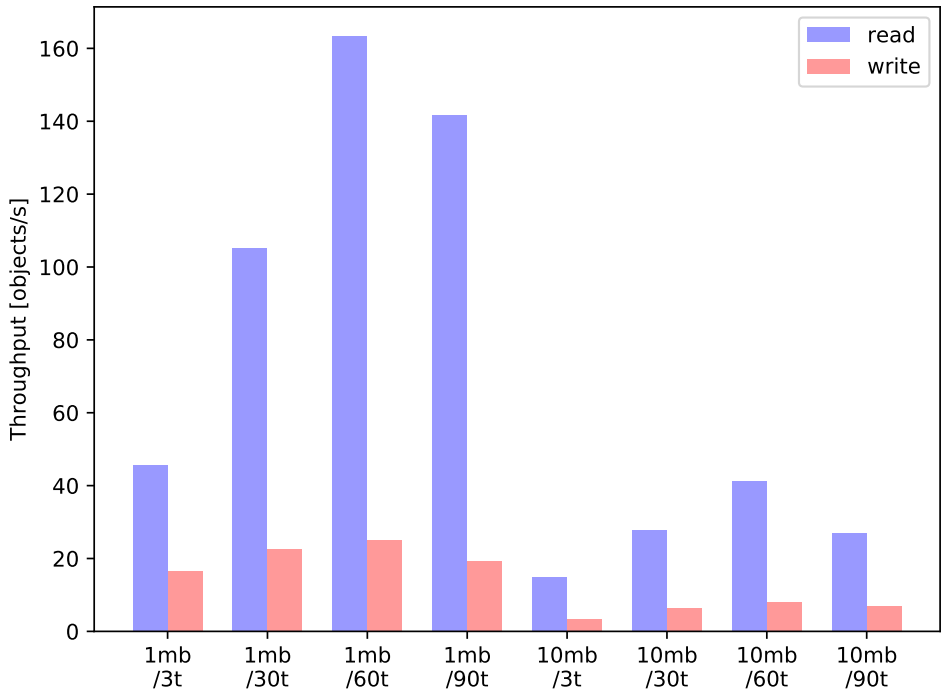

Fig. 6.5: Throughput of object storage system using implemented plugin dependent on number of threads. Object sizes 1MB and 10MB.

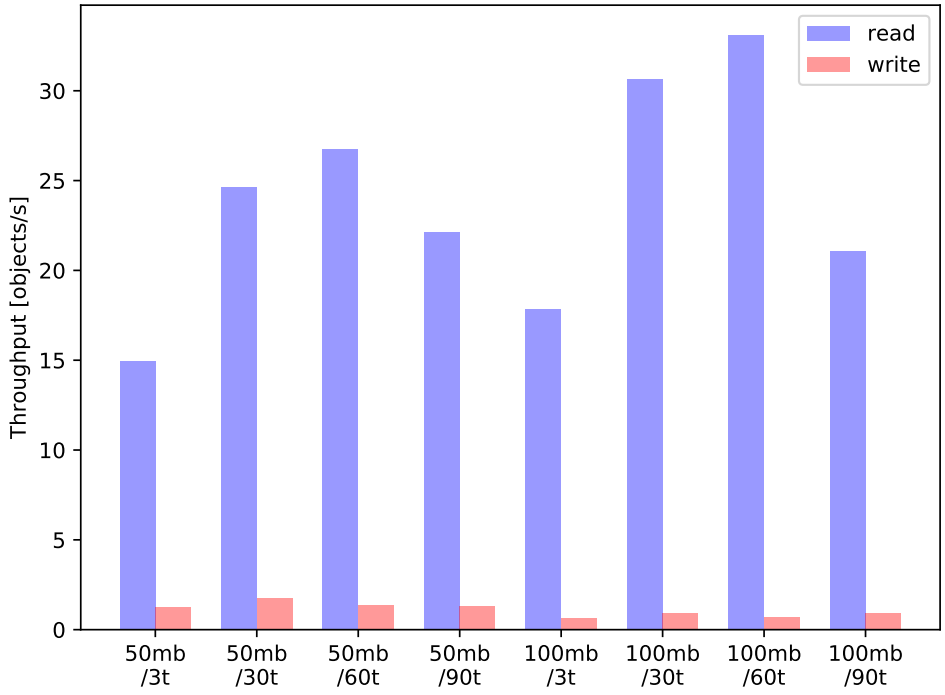

Fig. 6.6: Throughput of object storage system using implemented pluginindependent on number of threads. Object sizes 50MB and 100MB.

CHAPTER

SEVEN

CONCLUSION

This chapter summarizes and concludes the thesis.

LIST OF FIGURES

LIST OF TABLES

LISTINGS

BIBLIOGRAPHY

[s3d] Amazon s3: developer guide, working with amazon s3 objects, object key and metadata. URL: https://docs.aws.amazon.com/en_en/AmazonS3/latest/dev/UsingMetadata.html.

[Cepa] Ceph architecture. URL: http://docs.ceph.com/docs/master/architecture/.

[Cepb] Ceph block device. URL: http://docs.ceph.com/docs/master/rbd/.

[Cepc] Ceph filesystem. URL: http://docs.ceph.com/docs/master/cephfs/.

[ior] Ior interleaved-or-random. URL: https://github.com/hpc/ior.

[S3I] Introduction to amazon s3. URL: https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel.

[MSA] Managing concurrency in microsoft azure storage. URL: https://azure.microsoft.com/en-en/blog/managing-concurrency-in-microsoft-azure-storage-2/.

[min] Minio: high performance distributed object storage server. URL: https://www.minio.io/.

[Cepd] New in luminous: bluestore. URL: https://ceph.com/community/new-luminous-bluestore/.

[YCS] Yahoo! cloud serving benchmark. URL: https://github.com/brianfrankcooper/YCSB.

[lib] Libs3: amazon s3 library. URL: https://github.com/bji/libs3.

[HDF Group12a] *HDF5 User's Guide*. The HDF Group, 1.6.10 edition, 11 2012. URL: https://support.hdfgroup.org/HDF5/doc1.6/UG/.

[HDF Group12b] *HDF5 Virtual File Layer*. The HDF Group, 2012. URL: https://support.hdfgroup.org/HDF5/doc/TechNotes/VFL.html.

[HDF Group17] *The HDF Group. HDF5: API Specification Reference Manual*. The HDF Group, 2017. URL: https://support.hdfgroup.org/HDF5/doc/RM/RM_H5Front.html.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, 7–. New York, NY, USA, 2000. ACM.

[Cha14] Mohamad Chaarawi. User guide for developing a virtual object layer plugin. 2014.

[CK14] Mohamad Chaarawi and Quincey Koziol. Rfc: virtual object layer. 2014.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, 143–154. New York, NY, USA, 2010. ACM.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[MBT+12] Kshitij Mehta, John Bent, Aaron Torres, Gary Grider, and Edgar Gabriel. A plugin for hdf5 using plfs for improved i/o performance and semantic analysis. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, 746–752. Washington, DC, USA, 2012. IEEE Computer Society.

[WBMM06] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. Crush: controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06. New York, NY, USA, 2006. ACM.

[WLBM07] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In *Proceedings of the 2Nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing '07*, PDSW '07, 35–44. New York, NY, USA, 2007. ACM.