Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Analyzing Convergence Opportunities of HPC and Cloud for Data Intensive Science

Dissertation
zur Erlangung des akademischen Grades
Dr. rer. nat.

an der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim Fachbereich Informatik von

Frank GADBAN

Dezember 22, 2022

Gutachter:
Prof. Dr. Thomas Ludwig
Jun.-Prof. Dr. Michael Kuhn

Datum der Disputation: Dezember 05, 2022

Abstract

With the advent of the exascale era, the exponential growth in data volumes, and the rapid development of networking/cloud technologies, the cloud and HPC convergence is the subject of many conversations within the scientific community. In particular, HPC and cloud storage come from different assumptions that led to different underlying storage architectures and optimization techniques, which seem to be incompatible with one another from an abstract perspective. However, it is mandatory to overcome these differences and converge on the architectures so that scientific workflows do not need to differentiate between HPC and cloud storage but can benefit from the advantages of both.
This thesis investigates the HPC-Cloud convergence in the broader sense and focuses on the usage of cloud storage infrastructure for HPC workloads to optimize the scalability, performance, and cost-efficiency of the underlying infrastructure and improve the productivity of scientists running complex compute workflows.
In this work, the following research questions are addressed:
Can we use HPC and cloud storage technologies concurrently? What workflows will benefit from such settings, and which I/O interfaces are suitable? How can we achieve optimal data sharing between HPC and cloud resources? Is moving resource-demanding applications from on-premise to the public cloud a cost-effective solution compared to a hybrid alternative?
For this purpose, the term convergence is precisely defined, and a full-featured convergence assessment model is introduced and used to compare possible convergence scenarios. The comparison shows that using cloud storage inside HPC is one of the most promising approaches to leveraging HPC Cloud convergence. The performance of this scenario depends on the overhead introduced by using REST as a storage protocol in an HPC environment. A performance model for the relevant HTTP operations based on hardware counters is presented and experimentally validated. The obtained results reveal that an accurately configured REST implementation can provide high performance and match the HPC-specific implementation of MPI in terms of throughput for most file sizes and in terms of latency for file sizes exceeding one MB. Furthermore, the performance of the S3 interface offered by different object storage implementations on HPC and in the cloud is thoroughly investigated. The results indicate that the tested S3 implementations are not yet ready to serve HPC workloads directly, mainly because of the drastic performance loss and the lack of scalability. The approach to identifying the cause of the performance loss — by systematically replacing parts of the S3 stack — leads to introducing a new S3 access library, S3embedded, which proves to be highly scalable and capable of leveraging the shared cluster file systems of HPC infrastructure to accommodate several S3 client applications.

Using S3Embedded as a lightweight drop-in replacement for LibS3 is an enabling factor for Cloud-HPC agnostic applications that can be seamlessly executed in the public cloud or HPC and a massive step towards achieving HPC Cloud convergence.

Kurzfassung

Mit dem Aufkommen der Exascale-Ära, dem exponentiellen Wachstum der Datenmengen und der rasanten Entwicklung von Netzwerktechnologien ist die HPC-Cloud-Konvergenz Gegenstand wissenschaftlicher Diskussionen geworden.
HPC und Cloud-Storage gehen von unterschiedlichen Voraussetzungen aus, die zu verschiedenen zugrundeliegenden Speicherarchitekturen führen. Diese wiederum führen zu unterschiedlichen Optimierungstechniken, die aus einer abstrakten Perspektive nicht miteinander kompatibel zu sein scheinen. Es ist jedoch erforderlich, diese Unterschiede zu überwinden und die Architekturen einander anzunähern. Auf diese Weise müssen wissenschaftliche Arbeitsabläufe nicht zwischen HPC und Cloud Storage unterscheiden, sondern können von den Vorteilen beider Technologien profitieren.
Die vorliegende Arbeit untersucht die HPC-Cloud-Konvergenz im weiteren Sinne. Der Fokus liegt hierbei auf die Nutzung des Cloudspeichers für HPC-Workloads. Diese ist von Bedeutung, um die Skalierbarkeit, Leistung und Kosteneffizienz der zugrundeliegenden Infrastruktur zu optimieren. Darüber hinaus kann die Produktivität von Wissenschaftler:innen, die komplexe Computing-Workflows ausführen, verbessert werden.
In der vorliegenden Arbeit werden folgende Forschungsfragen geklärt:
Können wir HPC- und Cloud-Speichertechnologien gleichzeitig nutzen? Welche Arbeitsabläufe profitieren von solchen Einstellungen und welche I/O-Schnittstellen sind geeignet? Wie kann eine optimale Datenteilung zwischen HPC- und Cloud-Ressourcen erreicht werden? Ist die Verlagerung von ressourcenintensiven Anwendungen in die Public Cloud eine kostengünstige Lösung im Vergleich zu einer hybriden Alternative?
Um ein einheitliches Verständnis zu schaffen, wird der Begriff Konvergenz zunächst genau definiert und ein umfassendes Konvergenzbewertungsmodell eingeführt. Dies wird zum Vergleich möglicher Konvergenzszenarien verwendet. Der Vergleich zeigt, dass die Verwendung von Cloudspeicher innerhalb HPC einer der vielversprechendsten Ansätze ist. Die Leistung dieses Szenarios hängt von dem Overhead ab, der durch die Verwendung von REST als Speicherprotokoll in einer HPC-Umgebung entsteht. Es wird ein Leistungsmodell für die entsprechende HTTP Operationen auf der Grundlage von Hardware-Zählern vorgestellt und experimentell validiert. Die erzielten Ergebnisse zeigen, dass eine genau konfigurierte REST-Implementierung eine hohe Leistung bieten und mit der HPC-spezifischen MPI-Implementierung — für einige Dateigrößen — mithalten kann. Darüber hinaus wird die Leistung der S3-Schnittstelle gründlich untersucht. Die Ergebnisse zeigen, dass die getesteten S3-Implementierungen noch nicht bereit sind, HPC-Workloads direkt zu bedienen. Grund hierfür sind insbesondere die drastischen Leistungseinbußen und die mangelnde Skalierbarkeit. Die Hauptursachen für den Leistungsverlust konnten durch das systematische Ersetzen von Komponenten des S3 Stacks identifiziert werden. Dieser Ansatz wiederum führt zur Einführung einer neuen S3-Zugriffsbibliothek — S3embedded — die sich als hoch skalierbar erweist. Sie ist in der Lage, die gemeinsam genutzten Cluster-Dateisysteme der HPC-Infrastruktur zu nutzen, um mehrere S3-Client-Anwendungen unterzubringen.

Die Verwendung von S3Embedded als leichtgewichtiger Ersatz für LibS3 ist ein treibender und ermöglichender Faktor für Cloud-HPC-agnostische Anwendungen, die reibungslos in der öffentlichen Cloud oder im HPC ausgeführt werden können. Diese ist ein großer Schritt in Richtung HPC-Cloud-Konvergenz.

Acknowledgements

Throughout the writing of this dissertation, I have received a great deal of support and assistance.

Firstly, I would like to express my special thanks to my supervisor, Prof. Dr. Thomas Ludwig, for his support and guidance throughout my research. His expertise and insightful feedback pushed me to sharpen my thinking and bring my work to a higher level.

I would particularly like to single out Prof. Dr. Julian Kunkel: Thank you for the patient support, guidance, encouragement, and advice you gave me during my research. You also provided me with the insights and tools to choose the right direction and successfully complete my dissertation.

My heartfelt gratitude goes to my parents for their constant support and wise counsel.

I am especially thankful to my lovely wife and children for their understanding and ongoing support.

I would also like to extend my thanks to everyone at DKRZ who helped by providing the information or the resources needed for my research.

Lastly, I would like to thank all those who supported me directly or indirectly throughout this project.

"Denkst Du, das ist Luft, die Du gerade atmest?"

Morpheus

Contents

List of Figures

[1]Process per Node

List of Tables

Chapter 1

Introduction

At the time of writing, Exascale computing has become a reality[1], enabling breakthroughs in multiple scientific disciplines; Frontier is the first[2] system to hit the mark of one exaFlop. The possibility of using cloud technologies within such systems will definitely enhance the End-user experience and simplify complex and distributed workflows by expedicting the access to a wide range of resources. A seamless integration of supercomputing and cloud technologies is vital to allow organisations handle these unprecedented data growth rates. Although a subset of HPC[3] services are nowadays offered by public cloud providers, petascale data and computing capabilities and beyond are primarily provisioned within HPC data centers utilizing traditional, bare-metal resources to ensure performance, scalibility, and cost efficiency. Furthermore, on-demand and interactive provisioning of resources, which are common in cloud environments, continue to be difficult to achieve for most supercomputing ecosystems. This chapter gives an overview of HPC and Cloud concerning both computing and storage aspects. In Section 1.2.1 the benefits and drawbacks of HPC Cloud are exposed. HPC and Cloud convergence is currently an area of interest for both HPC and Cloud communities, fuelled to some extent by the conflict of interest between them; however, one of the main components of supercomputing, namely the highly performant storage I/O, seems to be overlooked. This aspect is reflected in Section 1.3, which represents the motivation behind this thesis. In addition, the goals of this work are shown in section 1.3.1. Finally, an outline of this thesis is laid out in section 1.4.

1.1 HPC

High-Performance Computing (HPC) utilizes clusters of powerful and fast interconnected computers to efficiently handle complex and data-intensive computational problems. These systems are managed by batch schedulers (Ma, Zhang, and Li, 2004) where user jobs are queued to be completed, based on resource usage and availability and without any visibility or concerns regarding the costs of running those jobs.

Applications that can efficiently be performed in parallel on different and highly interconnected nodes are considered typical HPC applications. They consist of parallel workloads or tightly coupled workloads.

- Parallel workloads are computational problems divided into small, simple, and independent tasks that can be run simultaneously, usually with little or no communication among them. Risk simulations, medical assessments, and logistics simulations are common examples.

[1]`https://top500.org`

[2]`https://spectrum.ieee.org/exascale-supercomputing`

[3]High Performance Computing

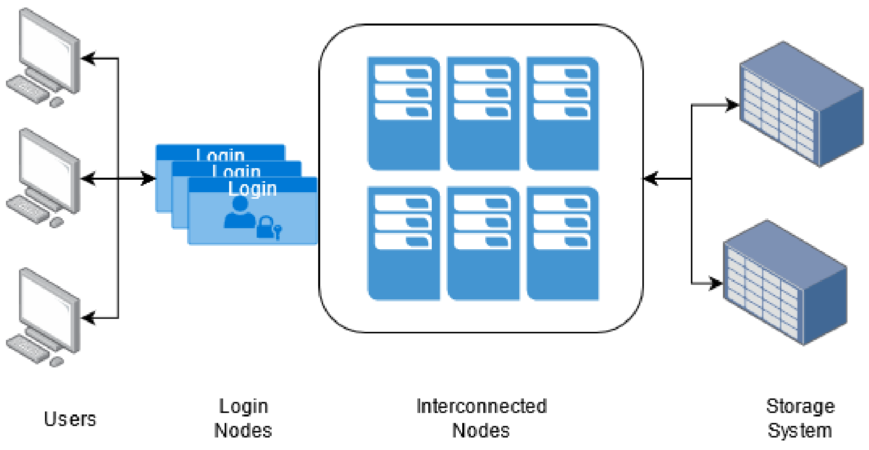


FIGURE 1.1: A typical HPC infrastructure

- Tightly coupled workloads typically take a considerable shared workload and break it into smaller tasks that interact continuously. As such, the cluster nodes communicate with one another as they perform their processing. These include weather forecasting, analyzing seismic waves, oil and gas exploration, quantum mechanics, fluid dynamics, simulating autonomous driving models, and product design.

These applications benefit from scale-up and scale-out performance: "scaling up" in the sense of using more powerful nodes/resources to process the workload or "scaling out" by just adding more nodes/resources to handle the workload, which is actually limited by the size of the HPC Cluster. The major HPC components are shown in fig. 1.1: compute, network and storage are typically "leading-edge technology" formed together to achieve high performance, high availability, and great speed of execution.

HPC storage usually consists of physical devices to manage, store and save the data and filesystem servers committed to running the file system interfacing with the application running on the HPC cluster. While the compute (CPU, GPU) and interconnect components are evolving at a high rate, the storage is still lagging [4]. The HPC Storage ecosystem will be explored in detail in chapter 2.

1.2 Cloud

The National Institute of Standards and Technology (NIST) definition (Mell, Grance, et al., 2011) seems to be the most prevailing within the academic community. It states that “cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction”.

Cloud computing includes both the applications provided as services over the Internet and the hardware resources and software stack needed to provide those services.

The main offerings of cloud computing can be summarized as follows:

- The impression of infinite computing resources available on-demand, as such, cloud computing users do not have to plan far ahead for provisioning;

[4]`https://top500.org`

- An up-front commitment is not required from cloud users, thereby companies are able to start small and increase hardware resources based on their needs; and
- The possibility to pay for using computing resources only when required and release them once finished, without any long-time commitment.

The profitability of a pay-per-use solution depends on many factors like the unit cost of cloud services in comparison to the dedicated, owned capacity, and the degree of usage of resources. Generally speaking, for most real-world workloads, leveraging a mix of on-premises and cloud-based capacity is likely to reduce cost vs. an on-premises-only approach.

Cloud storage is just one of the services offered by a cloud provider. It enables the central storage of data on a virtual hard disk on the Internet. One of the main benefits of cloud storage is its scalability. The required storage space can be expanded or reduced as needed.

A significant advantage of cloud storage is the exchange of data between scientists, as the cloud provides the ability to access files with any Internet-enabled device from any location. In this context, access rights, encryption authentication, and search mechanism should be guaranteed. The impact of the cloud model (private, public, hybrid) on security and costs should also be considered. A possible use case here is an internet accessible archive (for example, the Earth System Grid Federation, ESGF (Cinquini et al., 2014)) where users can download data to external sites or potentially process the data directly on-premises or in the cloud by deploying their analysis environments on a cloud platform that has direct access to the data.

As such, more and more scientists are using cloud storage mainly to share the obtained results as well as for archiving purposes. Direct processing of the data is also possible using services like HSDS (HDF-Group, n.d.).

On the other hand, the storage I/O is still considered a bottleneck in cloud environments (Yamato, 2016). While other researchers tackled the network performance (Persico, Montieri, and Pescape, 2016) of a cloud file system like Amazon S3 or evaluated the HPC applications on a cloud platform (Salaria et al., 2017) (Zhao et al., 2014), a relative few number (Kimpe and Ross, 2014; Matri et al., 2017) of researchers analyzed the performance of Intensive Data-driven applications on an HPC platform using cloud storage solutions.

In recent years, many cloud providers like (AWS, n.d.[a]) and (IBM, n.d.) started to offer HPC cloud services and regularly expanded them. Faced with tight budgets and non-sustainable projects, many research institutes and commercial HPC users take advantage of this cloud offering. Section 1.2.1 will thoroughly examine this new trend.

1.2.1 HPC Cloud

The popularity of cloud computing and the ease of resource allocation led to its adoption by several scientists. This, in turn, paved the way to the emergence of *HPC Cloud* - a.k.a. HPC in the Cloud - where cloud providers offer high-end hardware platforms and software environments to run HPC applications. Although this approach offers the advantages mentioned in Section 1.2 like immediate scalability, either horizontally or vertically, and the infinite cloud storage space for storing the results, it faces however many drawbacks, which can be summarized as follow:

- Performance problems: cloud providers tend to throttle network and storage I/O to allow, in the first place, a better sharing of the common resources and, in the second place, to enable granular billing of each IO Operation, this is without mentioning the virtualization overhead and the latency introduced by the network which is generally less performant that then one found in HPC

- Security Concerns: although cloud environments are nowadays encapsulated in a virtual private networking environment, and the connection to on-premises resources is protected with a VPN, running workloads in the Cloud faces many challenges like software vulnerability or zero-day attacks.
- Hidden costs: like the time spent finding the right cloud provider, the choice of high-end hardware to use, without mentioning traffic or just unseen costs, for example, when forgetting to delete unused virtual machine disks.
- Storage limitation: moving data between cloud and the on-premises data center is a costly and time-consuming process, due mainly to the large amount of data processed by typical HPC jobs.

What is needed here is a change in the way of thinking. The best solution is not to move the HPC workload to an HPC simulated environment in the cloud, but rather to think in both directions and find a way to move jobs seamlessly between HPC and cloud. This is not to be confused with cloud bursting, where workloads are dynamically offloaded to the cloud, based on specific criteria like the saturation of internal resources. To achieve this *Hybrid HPC* environment, a standard storage interface that can be used in HPC as well in the cloud is needed, and this is precisely what we will be exploring in this work.

1.3 Motivation

With the advent of the exascale era, the exponential growth in data volumes, and the rapid development of Networking/Cloud technologies, the convergence of HPC and cloud is the subject of many conversations within the scientific community. Nevertheless, an exact definition of the term convergence is needed since it seems to be understood differently. Furthermore, by adopting cloud computing, many companies seem to gain from cost-effective and flexible services to perform their data center workloads; however, whether cloud concepts or services are economically meaningful in an HPC environment still needs to be proven.

Most of the work addressing HPC and Cloud convergence focused on moving some processing tasks to the cloud or offering cloud-bursting (Lafayette, 2018), which means a data center may execute applications on the cloud when the waiting time for job execution in the data center exceeds a certain threshold due to high user demand. However, the most critical aspect to address is the **storage** aspect, responsible for the **data access, transfer, and manipulation** in a converged HPC Cloud environment, since it significantly affects the performance, cost, and security:

- **Performance**: The performance of using cloud storage inside HPC still needs to be evaluated. Many prerequisites must be fulfilled to consider cloud storage as an alternative to classic HPC storage, where low-latency, high throughput, high availability, highly parallel I/O, and high-scalability (Devresse and Furano, 2014) are expected. On the other hand, despite the considerable advancement in the networking capabilities, moving data between the cloud and the on-premises data center, or even between different regions/zones within the same cloud provider, is still a relatively slow process. This should be carefully evaluated, knowing that typical HPC workloads produce enormous amounts of data, as seen in chapter 2. Climate, weather, bioinformatics, and astronomy applications are a few examples of these workloads.

- **Cost**: As already noticed in section 1.2.1, running workloads in the cloud introduces the need to carefully assess which instance type to use, where to run these workloads, and most importantly, what storage type to use. Most cloud providers nowadays offer a limited SSD ephemeral storage coupled to each instance; the price of this "local" storage is included within the cost of running/leasing the instance. However, the cost of shared storage is relatively highly priced since it's IO-bound, in the sense that if more IO performance is required, the higher the price is. This can lead to exploding costs when running HPC workloads which usually require highly parallel I/O capabilities. The cost of moving the data should also be carefully examined: most cloud providers offer free ingress data transfer, i.e., when moving the data to the cloud, but charge for egress data transfer, i.e., when moving data outside the cloud to an on-premises data center or to another provider. This should be thoroughly evaluated, knowing that typical HPC Workloads generate vast amounts of data.
- **Security**: Security of data saved in the public cloud plays a significant role. Many issues must be addressed like access control, management of the encryption keys, breach notification, users' data privacy regulation, data lifecycle management, alongside the resiliency of the encryption system. This high level of security requires extra resources and negatively impacts performance.

As such, this thesis investigates the convergence between HPC and Cloud in the broader sense while focusing on the **storage** aspect in order to foster the usage of cloud storage infrastructure for HPC computing while optimizing scalability, performance, cost-efficiency, and productivity of scientists running complex compute workflows.

1.3.1 Goals

Our goal is to answer several high-level questions for research, for example:

- What defines a converged system and how to assess the level of convergence?
- Can we use HPC and cloud storage technologies concurrently? What workflow will benefit from such settings, which I/O interfaces are suitable?
- How can we achieve optimal data sharing between HPC and cloud resources?
- Is moving I/O demanding applications from on-premises to the public cloud a cost-effective solution compared to a hybrid alternative.
- To what extent can we use cloud storage in an HPC environment, what overhead to expect and how can we minimize this overhead?

Therefore, this work aims not only to measure performance but also to investigate new methods that allow the harnessing of the capabilities of both HPC and Cloud technologies. This will lead to an ultimate HPC Cloud convergence where scientific workflows do not need to differentiate between HPC and cloud storage but benefit from the advantages of both. These research topics are methodically investigated, and solutions are proposed and explored for relevant real-world scientific experiments. Appropriate scenarios are defined to address the issues of latency/bandwidth and to ensure technical feasibility; in this context, data interfaces and architectures are proposed, and several applications such as the concurrent use of heterogeneous storage systems are explored.

1.4 Outline of this Thesis

This chapter provided a brief overview of HPC, Cloud, and HPC Cloud concepts. The motivation behind this work and the aspired goals were also presented. In chapter 2, many terms and technologies relevant to this work are shown, like the different storage technologies and interfaces; a comparison matrix between those technologies is also provided; crefChapter2 also outlines the HTTP protocol evolution and describes the various convergence scenarios. In chapter 3, an assessment model is proposed to evaluate the different convergence solutions and conclude which scenario is the most promising. The assessment model also includes an economic decision model to help assess cost changes when bursting or moving entire workflows into the cloud. A real-world example is also illustrated.
In chapter 4, based on the results obtained in chapter 3, a comprehensive investigation of the overhead of the REST[5] protocol when using cloud services for HPC storage is presented, in particular, HTTP[6] is compared with an HPC native protocol, MPI[7]. The evaluation shows that REST can be a viable, performant, and resource-efficient solution, particularly for accessing large files. In chapter 5, the performance of the popular S3[8] cloud storage API[9] is assessed using specific HPC benchmarking tools in an HPC environment. Based on chapter 5 findings, chapter 6 introduces an alternative S3 library, S3Embedded, to leverage commonly shared file systems within HPC and accommodate S3 compatible client applications. It also assesses several improvements of the S3Embedded library by using the latest enhancements in the field of HTTP communication and suggests possible scenarios relevant for the climate and weather applications involving the simultaneous use of cloud and HPC storage systems. Finally, chapter 7 summarizes this thesis and outlines future work.

[5]REpresentational State Transfer
[6]Hypertext Transfer Protocol
[7]Message Passing Interface
[8]Simple Storage Service
[9]Application Programming Interface

Chapter 2

Background: Related Work & State of the Art

In this chapter, relevant terms and concepts for the following chapters and a detailed overview of the storage landscape in HPC and the cloud are provided. In section 2.3 the HPC relevant storage interfaces are illustrated, and the reasons causing the HPC community to relax the POSIX semantics are outlined.Section 2.5 summarises the evolution of the HTTP protocol. The relevant storage technologies used in HPC and cloud are discussed, and a detailed comparison is provided. Finally section 2.7 describes the different HPC Cloud convergence scenarios

2.1 Parallel Processing and Supercomputing

Parallel processing is the processing of a program with more than one execution unit. This term is multifaceted because parallel processing occurs at many levels of a parallel computer. Today's processors allow multiple execution units to execute more than one instruction in one clock cycle. There is support for quasi-parallel execution of multiple threads sharing the diversity of execution units in multithreaded processors such as the Intel Pentium Hyper-Threading. In a multicore processor, there are several cores in a processor chip, whereby each of these cores can, in turn, be a superscalar multithreaded processor, i.e., a processor that can execute more than one instruction within a clock cycle by simultaneously sending multiple instructions to distinct execution units on the processor (Farber, 2011). It is also possible to have different multicore processors on one motherboard. These boards or multiprocessors are interconnected to form a cluster. Numerous clusters can work together to create a cluster grid where parallel programs are executed.

High-performance computing or HPC, already described in chapter 1, uses high-end computers and parallel processing techniques to solve complex computational problems and promptly handle vast amounts of data.

2.2 Cloud Computing

The terms cloud and cloud computing were defined in chapter 1; in the following section, we introduce the different cloud types.

2.2.1 Cloud Types

Depending on where they are located, we can differentiate between three types of clouds:

- **Public cloud**: represents the large commercial cloud providers, such as Amazon Web Services (AWS), Google or Microsoft Cloud, which provide several services such as IaaS[1], PaaS[2], and SaaS[3] for many customers. Clients will be sharing in a "virtually" isolated way the computing, storage, and networking resources.
- **Private cloud**: usually found on-premises, i.e. within the organization's physical location and used to provide computing services and resources to the company's employees or its partners. It uses similar technologies as the ones used in the public cloud and is sized to meet the specific organization's needs. Since the total number of resources is limited in this case, the use of smart scheduling is essential to efficiently allocate the private cloud resources to the applications that need them while respecting the deadline set for those applications. An HPC datacenter can, to some extent, be compared to a private cloud providing needed resources to students and researchers from the universities and entities who contributed to set up the data center.
- **Hybrid cloud**: In this case, the cloud infrastructure consists of different cloud infrastructures put together to form a hybrid cloud environment. Although these clouds are kept separate but can be consolidated to accommodate particular applications at defined times. A typical use case is when the private cloud requires extra capacity, it would "burst" some workloads to a different cloud. This will be discussed further in section 2.7.1. Usually caused by approaching deadlines for resource-intensive applications, the ability to seamlessly extend HPC applications' computing resources by bursting to the public cloud is an essential consideration, notably as more organizations set up cloud-computing environments (Linthicum, 2016).

2.2.2 Cost Comparison: On-premises vs Cloud

The cost comparison between running workloads on-premise or in the cloud has been the main subject of many research papers (Smith et al., 2019; McGough et al., 2014; Emeras et al., 2016); the majority of them concluded that, despite the marketing done by the major cloud providers, the cost for running scientific workloads on-premises is cheaper than running them in the cloud.

On the other hand, in (Weinman, 2011), the author concluded that *"a pay-per-use solution makes sense if the unit cost of cloud services is lower than dedicated, owned capacity. Clients can save money by substituting fixed infrastructure with clouds when workloads are spiky, specifically when the peak-to-average ratio is greater than the ratio of the cloud vs. fixed resources on a unit cost basis"*. He concluded that leveraging a mix of on-premises and cloud-based capacity for most real-world workloads is likely to reduce cost versus an on-premises-only approach.

He noted that sharing with other tenants can save on common costs, like the staff cost to take care of the whole infrastructure, expertise, and security that is hard to find within small organizations. However, as seen in section 3.4, the move to the cloud requires another set of skills that might be even harder to find nowadays since those types of skills are highly demanded. (McGough et al., 2014) mentioned as well that once a critical mass of the HPC platform is reached, the total cost of running scientific workloads is mainly in favor of the on-premises cluster.

[1]Infrastructure as a Service
[2]Platform as a Service
[3]Software as a Service

In chapter 3, the main factors involved in the cost calculation for both scenarios are thoroughly examined; furthermore, a real-world example is provided.

2.3 Storage Interfaces

Storage interfaces provide the communication point between the application and the file and storage system. They should be suitable and easily usable, allowing the developers to focus on the applications' functionality instead of handling the I/O interface. In the HPC context, they must deliver high performance while supporting parallel access. Furthermore, the ability to exchange and share data efficiently is one of the most significant points to address in research environments. The following sections will describe standard storage interfaces found in the HPC world.

2.3.1 POSIX-IO

Initially, the name "POSIX" referred to the IEEE Std 1003.1-1988, released in 1988 to resolve portability issues. The latest version, at the time of writing, is IEEE Std 1003.1-2017.

POSIX does not define the operating system; it only describes the interface between an application and an operating system. Its purpose is to enhance portability by setting a set of standards to follow. The full POSIX standard is 4000-plus pages with more than 1350 interfaces.

The POSIX I/O standard offers a model for file system organization by grouping the files into directories. As seen in (Todd, 2017), it defines numerous functions to manipulate the data:

- mount, umount the file system
- open, close file descriptor
- write, read to/from file descriptor
- mkdir, rmdir, creat, unlink, link, symlink
- fcntl (byte range locks, etc.)
- stat, utimes, chmod, chown, chgrp

Several strict consistency semantics are defined, ensuring that all accesses are atomic operations. For example, writing to a file must appear as one atomic operation to any readers accessing the file during the write.

(Todd, 2017) states also that *"POSIX file systems are treated as a sequence of bytes. However, internally the data content of a file is stored as logical sequence of file system blocks: Each block is a fixed number of bytes. The last block might not be full. Within a block, all the bytes are sequential. However, within a file, the blocks might not reside sequentially on the disk. Underlying storage systems are usually organized as blocks, and ideally, file system blocks are aligned with the storage system's blocks. File system refers to the info about the data as metadata; an inode is defined for each file or directory to provide the locations of the data blocks for the file and the file attributes, like "time last accessed" or "owner of the file". The location of each inode is found in the inode table"*.

The POSIX-IO API also defines how the data and metadata of files are cached, reducing the cost of precisely tracking access, update times, and the file size. It is worth noting that POSIX was developed with a single system approach in mind, at

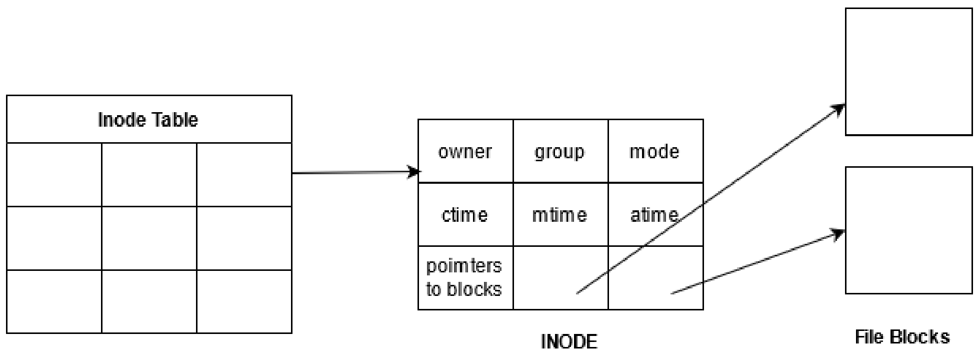


FIGURE 2.1: POSIX Data Blocks

a time where a single computer operating system managed its local file system, and concurrent access were only restricted to the processes running on that operating system. In this case, all file operations we accomplished on a single device, and the atomic access to files was achieved using locks. However, in a highly distributed or parallel file storage environment, shared between multiple clients and processes, the act of maintaining those semantics proved to be a complex and communication-intensive process.

For example, when fetching a list of files in some directory, as seen in listing 2.1, the readdir function retrieves only the file names, although, to get more information, like the file size, a stat request needs to be made for each file found, resulting in a network call for each of those files. On the other hand, caching is no longer straightforward since remote invalidation is needed to keep cache contents consistent. As such, the cost of presenting a single global, uniform view of a subset of the file system becomes prohibitive.

LISTING 2.1: List Files

```
/*
 * to display the names and sizes of all files in the current directory.
 */

#include <dirent.h>
#include <stdio.h>
#include <sys/stat.h>

int main(void)
{
  DIR *d;
  struct dirent *dir;
  d = opendir(".");

  struct stat stats;

  if (d == NULL)
  {
    printf("Could␣not␣open␣current␣directory");
    return 0;
  }
  while ((dir = readdir(d)) != NULL)
  {
    printf("\nFile␣NAME:%s", dir->d_name);

    if (stat(dir->d_name, &stats) == 0)
    {
      // File size
      printf("\nFile␣size:␣%li", stats.st_size);
    }
    else
    {
      printf("Unable␣to␣get␣file␣attributes.\n");
    }
  }
  closedir(d);
  return (0);
}
```

"The most common approach to implementing these semantics is to utilize a locking subsystem to control access to files, typically found in numerous systems that enforce the POSIX semantics like GFS, GPFS, and Lustre" (Sterling, 2002).

A process that needs to write to a file section must first obtain the lock associated with that section. After writing, it releases the lock. Advanced lock caching alleviates the locking subsystem's overhead in systems presenting a high level of simultaneous access.

Filesystems may implement locks at the block, file, or extent granularity. File-based locks associate a single lock with a file. This approach is considered the least performant due to the competition for locks during simultaneous access. Block-based locks is commonly used in systems that require block-based access when communicating between clients and the underlying storage. Although this type of lock is much finer-grained than file-based locks, it might however lead to the creation of many locks, especially for large files. A workaround would be to increase the size of blocks, although this might result in false block sharing.

Extent-based locks presents also a flexible locking method. *"This method might result in fewer used locks since large ranges are described as a single extent. However, this benefit is lost if accesses are interleaved at a fine granularity. When associated with non-contiguous access, this procedure can also lead to a significant number of locks in the system. Despite these two disadvantages, this is the best locking approach for concurrent access under POSIX used by parallel file systems"* (Sterling, 2002).

Scientific access patterns are notably regular. However, none of this information is retained in any of the cited locking approaches, making them somewhat inefficient, either in the number of locks or in competition for a small number of locks. It is worth noting here that the POSIX semantics are known to be a problem in the community (Zadok et al., 2017). This problem is similar to the ones encountered in distributed shared memory (DSM) systems, where hardware and software construct globally accessible memory regions.

POSIX HPC I/O extensions

As seen above, the cost of preserving the POSIX consistency increases considerably with the massive parallel access. It generates a metadata access bottleneck when creating or updating files concurrently or synchronously.

Distributed HPC processes typically produce concurrent and similar operations on many files. For example, suppose that many nodes need to open the same file. Despite that POSIX file handles are only valid on the local node, each node needs to cross the directory hierarchy to find the requested file, producing a large number of metadata requests and as such a high overhead on the remote file system. *"The POSIX HPC extensions attempt to decrease this load by allowing a single node to open the file and then export some representation of the resulting file handle (for example, a direct pointer to the enclosing directory) to other nodes (openg function), which then convert the exported handle directly to a file handle (sutoc function) without having to perform a full open call"* as described in (Kimpe and Ross, 2014).

The main goal of the HPC I/O extensions is to create a standard way to provide high performance and good semantics. The primary approach is either to relax semantics considered expensive or to provide more information to inform the storage system about access patterns.

For example, the lazy I/O data integrity concept already found in NFS can be implemented by specifying O_LAZY in flags argument to open(), allowing the network filesystem to relax data coherency requirements to improve the performance of shared-write access. Other client processes are unaware of writes until one of the following calls is issued: lazyio_propagate(), fsync(), or close(). On the other hand, the system can cache the accessed data till lazyio_synchronize() is requested.

The filesystem does not ensure synchronization across processes or nodes; the application should use external synchronization methods (e.g., pthreads, XSI message queues, MPI) to achieve the required consistency.

Although the POSIX HPC I/O extensions aim to address some of POSIX's limitations, however, none of the extensions are integrated into any major file system.

POSIX-IO models

In network programming, a socket is used to transmit and receive information through a network; it is a particular file that can be compared to a pipe. It has two ends; usually, a server and a client that might reside on the same or two different machines. Writing data to a regular file is only possible as long as the container (file system - or - quotas) allows it. Data goes into a buffer, and then that buffer is transported over the network. We will distinguish here between four common POSIX I/O models:

- Synchronous blocking I/O (Synchronous, blocking I/O)
- Synchronous non-blocking I/O (Synchronous, non-blocking I/O)
- I/O Multiplexing (I/O Multiplexing)
- Asynchronous I/O (Asynchronous I/O)

Blocking I/O

The simplest model is the blocking I/O model. As seen in fig. 2.2 The client issues a recvfrom() system call to read the data from the remote system. Meanwhile, the caller process is blocked until the kernel receives the data and copies it to the process before returning. After encapsulation, the network card will pass the data message to the protocol stack and copy it to the kernel buffer. Once the data is ready and arrives in the kernel buffer, the kernel will copy the data to the user memory space and return the result, freeing the buffer. At this time, the caller process will unblock and resume execution. Blocking I/O does not waste CPU time slices but can only handle one connection at a time.

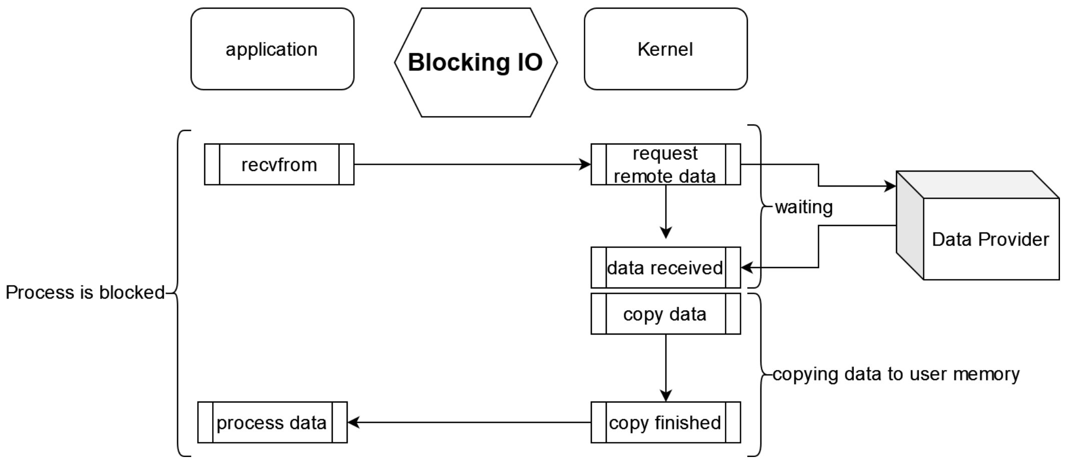


FIGURE 2.2: Blocking IO

Non-blocking I/O

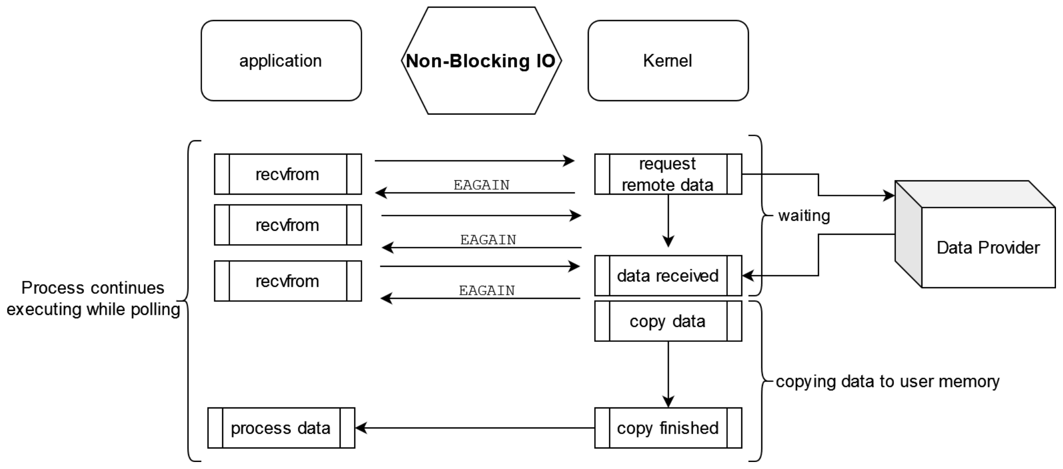


FIGURE 2.3: Non-Blocking IO

Unlike blocking I/O, non-blocking I/O means that when reading data using recvfrom(), and so long the data is not ready, an error is returned immediately without that calling process gets blocked, allowing it to continue executing other operations. To read the data, it will continuously call the recvfrom() polling operation. Once the data is ready, the kernel will copy the data to the user memory space and return the result of successful reading. The disadvantage of this model is that polling operations will take up time slices and waste CPU resources.

IO Multiplexing

I/O multiplexing is the capability to perform simultaneous I/O operations on multiple file descriptors. The select() system call is used to perform multiplexing; it returns the number of ready descriptors. A call to select() blocks the calling process until the given file descriptors are ready or until an error or timeout occurs. The process can do I/O on these file descriptors before the next iteration of the I/O multiplexing.

Figure 2.4 shows that this is quite similar to blocking I/O, with both phases blocking. But an essential difference is that it can wait for multiple file descriptors to be ready, i.e., it can handle multiple connections.

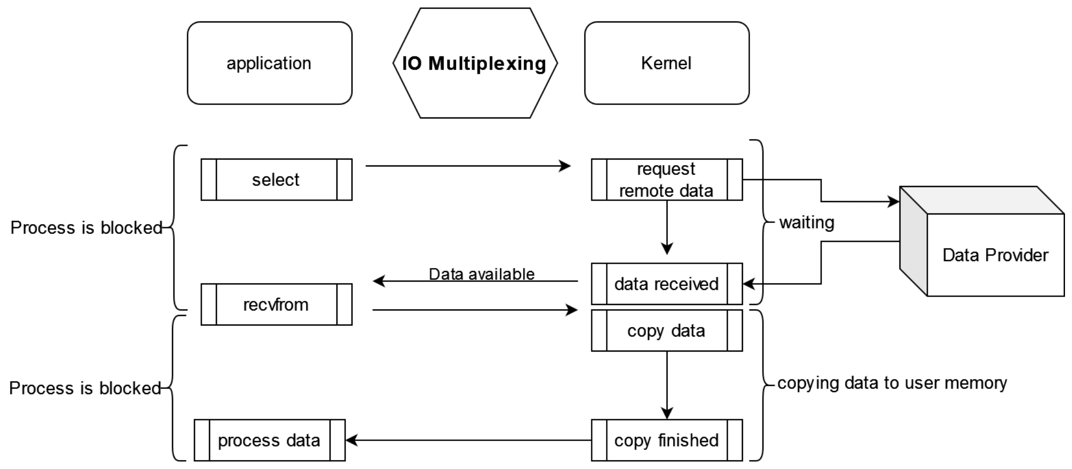


FIGURE 2.4: IO Multiplexing

This model can be extended to use multithreading with blocking I/O: rather than using select to block on different file descriptors, the program uses various threads, one for each file descriptor, and each thread can call recvfrom().

Note that the Linux OS introduced epoll() to overcome the shortcomings of select() like the low-level polling mechanism, which increases the overhead and the limited number of file descriptors.

Asynchronous IO

The semantics of the read operation is different from the above models. The read operation "aio_read" here will notify the kernel to perform the read operation, copy the data to the process, and inform the process after the completion of the entire operation, hence binding a callback function to process the data. The reading operation will return immediately, and the program can perform other operations. After completion, the process is notified, and the process calls the bound callback function to process the data.

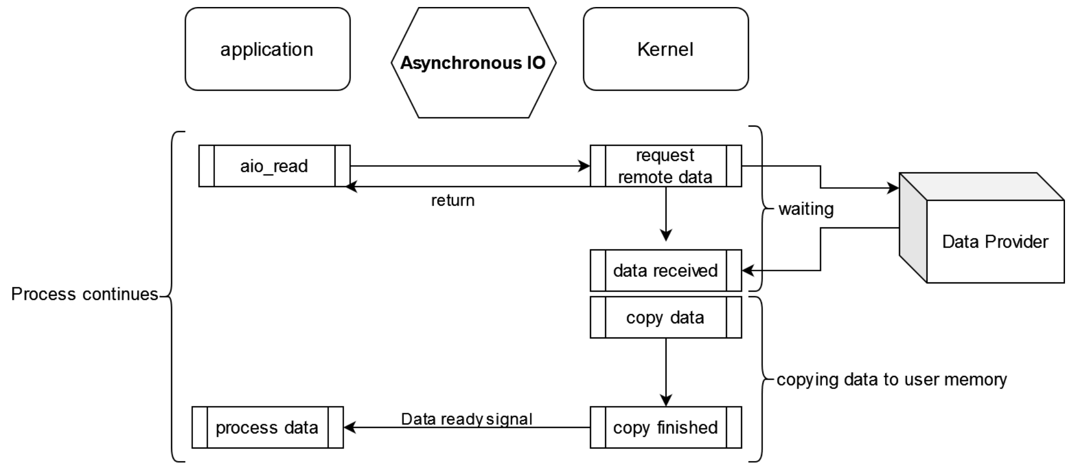


FIGURE 2.5: Asynchronous IO

Asynchronous I/O is rarely used in network programming but may be used in File I/O.

2.3.2 MPI-IO

The Message Passing Interface (MPI) is a message-passing standard used to program process communication running on parallel computing architectures. Version 2 (Forum, n.d.) of this interface introduced in 1997 the support for file I/O (MPI-IO). MPI-IO offers a file view to the MPI processes, representing wherein the file a process will be writing. As such, the act of writing data to a file is similar to sending a message to that file.

This can be seen in listing 2.2. Subsequently, reading and writing of non-contiguous slices of more complex data layouts, like large multi-dimensional arrays usually found in scientific computing, is very similar.

LISTING 2.2: MPI-IO Example

```
#include <mpi.h>
MPI_Status status;
MPI_File fh; MPI_Offset offset;
MPI_File_open(MPI_COMM_WORLD, "/pfs/datafile", MPI_MODE_RDONLY,
    MPI_INFO_NULL, &fh)
nints = FILESIZE / (nprocs*INTSIZE);
offset = rank * nints * INTSIZE;
MPI_File_read_at(fh, offset, buf, nints, MPI_INT, &status);
MPI_Get_count(&status, MPI_INT, &count);
printf("process␣%d␣read␣%d␣ints\n", rank, count);
MPI_File_close(&fh);
```

(Gropp, 2016)

The MPI-IO specification permits an application to write into separate files or into the same file from different MPI processes. It uses MPI datatypes to specify both the file and the process data. It supports collective IO, thus offering a high level of abstraction, allowing the smooth implementation of regular parallel operations like each process writing to its part of a file or collectively with its other processes.

In contrast to the instant visibility of changes for all involved processes as required by POSIX, the default semantics in MPI-IO ensure that the changes written by a certain process will be immediately visible to that process, although not to the other processes, they should first synchronize their view of the file using MPI_File_sync and MPI_Barrier (Corbett et al., 1996). The semantics are more relaxed as in POSIX I/O; nevertheless, since MPI-IO is commonly built on top of POSIX I/O functionality, it cannot deliver this flexible data consistency in every implementation.

2.4 Storage Access Technologies

2.4.1 Parallel File System

Within a parallel file system, data blocks are striped across multiple storage devices on numerous storage servers utilizing the resources of these devices in parallel. Increasing the available servers and storage devices generally improves the throughput, given that the underlying network bandwidth can support it. In this case, each sequential block I/O request can potentially go to a completely different server or storage device. Parallel file systems are a subset of clustered file system, since they can be shared or simultaneously mounted to multiple nodes. In the upcoming subsections we will briefly outline Lustre and BeeGFS, which power most of the systems submitted to the IO500[4], which is an I/O benchmark that measures HPC storage system performance, releasing an updated IO500 list twice a year. Other examples include the Spectrum Scale / General Parallel File System (GPFS) (Barkes et al., 1998), WekaFS [5] or GekkoFS[6]. other products that provide this sort of clustered parallel access to data. These products offer performance and capacity scalability as described below:

Lustre

Lustre (Braam, 2019) is one of the most used parallel distributed file systems on supercomputers. It is licensed under the GNU General Public License (GPLv2) and can, as such, be altered and enhanced. Lustre's architecture seen in fig. 2.6 differentiates between clients and servers.

"Clients use RPC messages to interact with the servers, which perform the actual I/O operations. While all clients are identical, the servers have different roles: Object Storage Servers (OSS) manage the file system's data represented in the form of objects; clients can access byte ranges within the objects. Metadata Servers (MDS) manage the file system's metadata; clients can contact the appropriate OSSs independently after retrieving the metadata. Each server is connected to possibly numerous targets (OSTs/MDTs) that store the actual file data or metadata, respectively. Lustre runs in kernel space, where most of its functionality has been implemented in kernel modules" (Lüttgau et al., 2018).

By using the kernel's virtual file system (VFS), Lustre is able to offer a POSIX-compliant file system. However, each file system operation leads to a system call and eventuel overhead when addressing high-performance network and storage devices.

BeeGFS

BeeGFS (Heichler, 2014) is a software-defined parallel clustered file system developed with a strong focus on performance and intended for straightforward installation and ease of management. It started as an in-house program at the Fraunhofer Center for HPC in 2005 and was initially perceived as the Fraunhofer filesystem. BeeGFS is built on highly efficient and scalable multithreaded core components with native RDMA[7] support. Like the Lustre file system, BeeGFS separates data services from metadata services; however, a significant improvement is the capability to thread the metadata request service using a built-in queue and define how many threads to spawn on each metadata server as needed. After the client obtains the metadata information from the metadata servers, it can immediately access the data. BeeGFS offers a straightforward approach; however, it does not natively support any kind of data protection such as erasure coding or distributed RAID. At the time of writing, this is possible when using 3rd party solutions, like NVMesh NVMe devices. BeeGFS claims to be suitable for I/O intensive workloads because of its parallelism. A BeeGFS based storage system is currently (2021) ranked 14 on the IO500.

[4]`https://io500.org/list/isc21/io500`

[5]`weka.io`

[6]`https://storage.bsc.es/gitlab/hpc/gekkofs`

[7]Remote Direct Memory Access

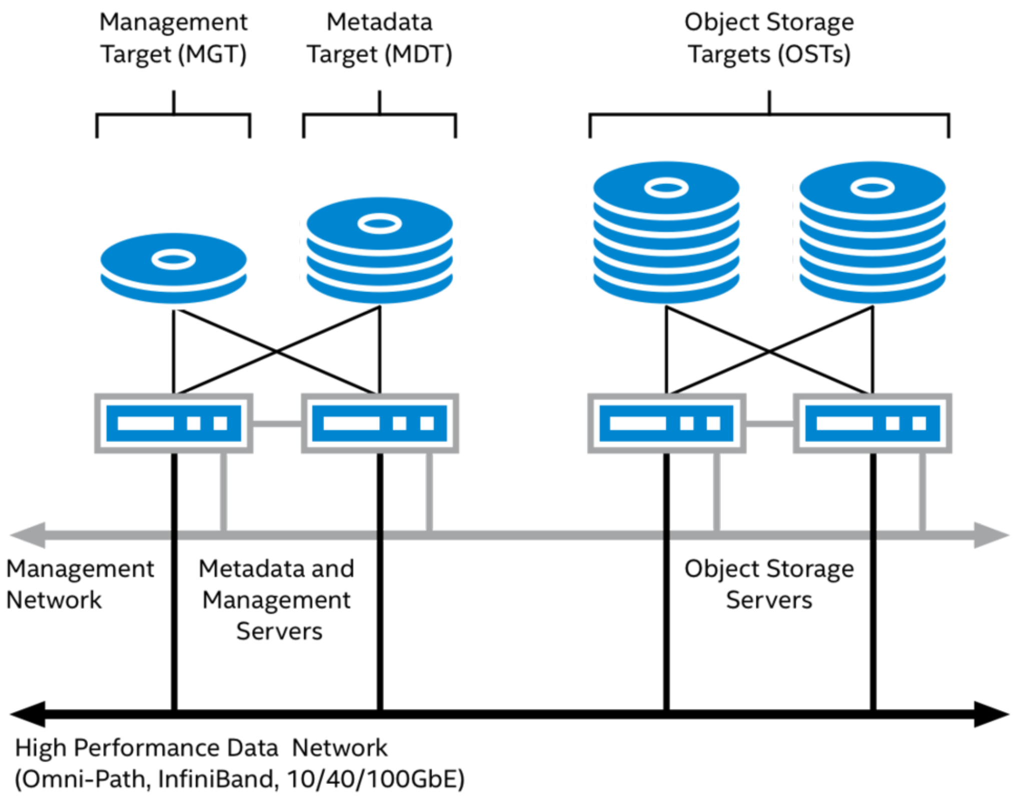

Management Target (MGT)
Metadata Target (MDT)
Object Storage Targets (OSTs)
Management Network
Metadata and Management Servers
Object Storage Servers
High Performance Data Network (Omni-Path, InfiniBand, 10/40/100GbE)

FIGURE 2.6: Lustre Architecture (Braam, 2019)

2.4.2 Object Storage

Object storage is a storage architecture that arranges information into containers of adjustable sizes, referred to as objects. Each object holds the data itself as well as its related metadata and has a unique identifier that is used to locate it, rather than a file name and path. Organizing these unique identifiers into a flat address space mitigates the complexity and scalability challenges of a hierarchical file system using file paths. An object can be found using this unique key independently of its physical location on a distributed system. Therefore, object storage is highly scalable by design and is essentially different from conventional block or file storage systems.

Cloud storage is a subset of object storage where objects are accessed directly from the client application, using a RESTful API (Richardson and Ruby, 2008).

In the world of HPC, computational performance has long exceeded the performance of the traditional file-centric storage systems since the POSIX file system interface was hardly suitable for data management on supercomputers (Zadok et al., 2017). Many workarounds were proposed to address this issue; some of them tried to introduce evolved I/O algorithms in MPI, like Data aggregation/sieving in ROMIO, (Thakur, Gropp, and Lusk, 1999) or to implement different data organizations on the back-end storage, like PLFS (Bent et al., 2009) or to introduce richer data formats, for example, HDF5 (Folk et al., 2011), NetCDF (Rew and Davis, 1990a).

Eventually, and although a file represents a convenient way to store the data, the ideal concept for scientific computing/HPC would be rather the use of a data object model (Liu et al., 2018) where all levels of metadata are encapsulated. This evolutionary path to Object Storage Access (Goodell et al., 2012) may lead ultimately to seamless integration with a cloud infrastructure. As we see nowadays, file systems like QuoByte and Ceph (Weil et al., 2006) export various file system APIs, including S3 (AWS, 2020). There are cases in which low-latency is particularly emphasized in the design of the HPC storage protocol, as in DAOS (Lofstead et al., 2016), for example. A provoking question for the future is, if, in the long run, a cloud API like S3 becomes similarly efficient, then why should we retain HPC-specific options? Will these remain important or just become a niche?

An object store holds the objects in a flat data environment without using folders, directories, or complex hierarchies as in a file-based system. Each object is a simple, self-contained container containing the data as well as its metadata; a unique ID is utilized to locate and access the object instead of a file name and path. High scalability is guaranteed by aggregating object storage devices into more extensive storage pools spread across different locations, gaining data resiliency and disaster recovery.

While some distributed file systems, like Lustre, mentioned earlier in section 2.4.1, use an object-based architecture, where file metadata is held on metadata servers, file data is stored on object storage servers, and the filesystem abstracts them to present a POSIX compliant filesystem view to users and applications, they do not provide direct access to the data files/objects using a predefined REST API [8].

To avoid any ambiguity, and for the rest of this work, we will only use the term object storage for the systems capable of offering a REST API to access the objects/files, i.e., for cloud storage.

Cloud Storage, like Amazon Web Services S3[9], Microsoft Azure Blob Storage, or Google Cloud Storage, are typical examples of object storage. The number of objects hosted by those systems depicts the vast scalability achieved by object storage; for example, Azure claimed in 2014 to host more than 30 trillion storage objects [10]

The following subsections outline some of the most common object storage systems.

Ceph

Ceph (Weil et al., 2006) is an open-source software capable of running on commodity hardware to offer a highly scalable object-, block- and file-based storage system. Ceph introduces the CRUSH (Controlled Replication Under Scalable Hashing) algorithm to ensure that the data is uniformly spread across the cluster and easily retrieved by all cluster nodes. As shown in fig. 2.7, Ceph object storage can be accessed using the OpenStack Swift or the Amazon Simple Storage Service (S3) APIs alongside a native API which can be used for direct integration with supported software applications. The CBD or Ceph Block Device is a virtual disk that can be attached to virtual machines or bare-metal Linux-based servers. The Ceph Reliable Autonomic Distributed Object Store (RADOS) ensures Block storage capabilities like snapshots and replication. RADOS can be used as a backend for the OpenStack Block Storage. Ceph file system (CephFS) is a POSIX-compliant filesystem.

MinIO

MinIO (MinIO-Inc., n.d.) is an open-source object storage server which gained a certain popularity in recent years since it offers an Amazon S3 compatible API. It is

[8]REST API for the Lustre Integrated Manager `https://github.com/whamcloud/Online-Help/blob/master/docs/api/rest_API.md`

[9]Simple Storage Service

[10]`https://www.neowin.net/news/microsoft-azure-by-the-numbers/`

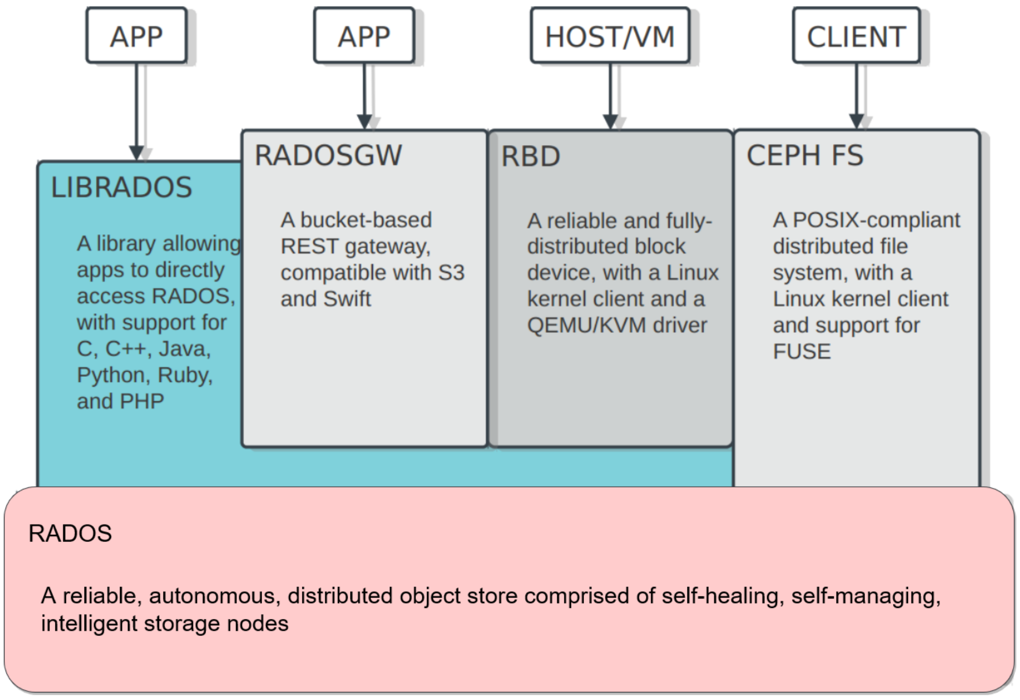

APP
APP
HOST/VM
CLIENT
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
RADOS
A reliable, autonomous, distributed object store comprised of self-healing, self-managing, intelligent storage nodes

FIGURE 2.7: Ceph Architecture (Weil et al., 2006)

conceived for storing unstructured data like pictures, videos, log files or data back-ups. Similar to Amazon S3, the maximum size of an object is 5 TB. AWS Lambda notification is also provided and its data protection abilities can endure a failure of up to half the number of servers and drives in use with its erasure code and bit rot detection that are able to detect corrupted data in container deployments.

A detailed overview of the different MinIO modes is covered in chapter 5.

Swift

The OpenStack Object Storage Swift (OpenStack-Foundation, n.d.) is an open-source software used to handle the storage of considerable amounts of data cost-effectively by using clusters of standard server hardware.

OpenStack Swift deposits the data as binary objects on the server operating system's underlying file system, with each object having its related metadata. Data and metadata are stored together and copied as a single unit.

The OpenStack Swift architecture shown in fig. 2.8 presents a proxy server and multiple storage nodes. The Swift REST-based API runs on the proxy server to communicate read and write HTTP requests between clients and the storage servers .*"The proxy server locates the objects by their hashtags and metadata, whereas usual HTTP methods like PUT and GET are used to store and retrieve these objects and their associated metadata from the Swift cluster. It also ensures the completion of writes to drives on the storage nodes. Replication and erasure coding are also offered across the storage nodes in the server cluster. It also introduces the concept of region and zone by placing each object in locations as unique as possible. If a server or hard drive fails, the Object Storage system replicates its content from running nodes to other locations in the cluster."* [11]

[11] https://www.techtarget.com/searchstorage/definition/OpenStack-Swift/

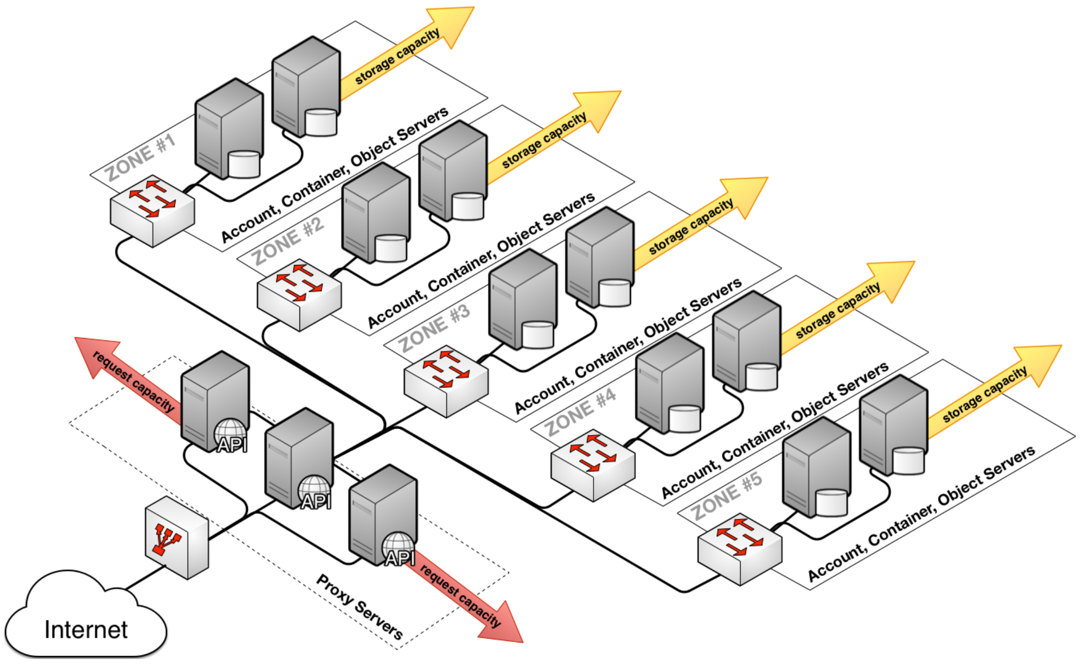


FIGURE 2.8: Swift Architecture (OpenStack-Foundation, n.d.)

	lustre	BeeGFS	minio	cephfs	swift	DAOS
License	GPLv2	GPLv2	AGPLv3	LGPLv3	Apache License 2.0	BSD-2
S3 support	no	no	yes	yes	yes	no
Scalability	high	high	low	medium	medium	high
Replication	yes	yes	yes	yes	yes	yes
Data Protection	yes	not native.	yes	yes	yes	yes
High Availability	yes	yes	yes	yes	yes	yes
POSIX Compliance	high	high	low	acceptable	relaxed	relaxed
File storage support	yes	yes	no	yes	no	no
Block storage support	no	no	no	yes	no	no
Central metadata servers	required	required	not needed	required	not needed	not needed

TABLE 2.1: Comparison matrix of the different filesystems

2.4.3 DAOS

DAOS (Lofstead et al., 2016) *Distributed Application Object Storage* is Intel's open-source and parallel file system for high-performance file system operations. It is optimized for memory interface Optane DIMMs and NVMe-accessed Optane and NAND SSDs. As such, no traditional magnetic hard-disks are found in a DAOS system. DAOS places metadata and stages small IO operations into Optane Persistent Memory on the same node as the block storage (NVMe drives) before writing entire blocks to the SSDs. The metadata in the DCPMM represents the data structures maintained by each DAOS server to reach the application data saved in DAOS containers and objects on the SSDs. As such, DAOS does not require any dedicated metadata servers or head nodes.

2.4.4 Comparison Matrix-Storage Solution

In section 2.4.4 several factors are used to compare the filesystems mentioned above, referring mainly to the published documentation for each of them.

- License: the licensing model can hint at the cost of licensing and using the product. It goes without a doubt that GPL and free software licenses are the most common.
- File storage support: where files are named, tagged with metadata consisting of the file name, file type, and its creation and update time, and organized in folders
- Block storage support: where a file is divided into equally-sized chunks of data called blocks stored separately under a unique address. A server operating system uses this individual address to pull the blocks together, assembling them into a file. These blocks can be stored anywhere in the system for maximum efficiency, sparing the time to navigate through a folder hierarchy to access the data blocks.
- S3 support: since AWS S3 (AWS, 2020) evolved to be the de-facto standard interface for accessing cloud storage, it is worth considering if the system offers an S3 interface.
- Scalability: Possibility to increase the data cluster when needed; without that, the performance suffers. To quantify the scalability, we consider the frequency of the occurrence of the respective filesystem on the io500 [12] list and the most extensive implementation of the system.
- High availability: where data is automatically replicated from one storage node to multiple other nodes. If a given data set in a given node gets compromised or deleted unexpectedly, the data can be copied back from the additional copies found in the same system.
- POSIX Compliance: To assess the POSIX compliance, we found that most file system implementations do not rigorously adhere to the POSIX spec; for instance, most filesystems relax the atomicity requirements for reads due to performance reasons. We found that the "Lustre code is not completely POSIX compliant" [13] and that "CephFS diverges from strict POSIX semantics" and it "relaxes more than local Linux kernel file systems (e.g., writes spanning object boundaries may be torn)."[14]. The authors of DAOS clearly state that they are shifting away from block-based, POSIX-compliant file systems towards more scalable, transactional object stores.

2.5 HTTP Evolution

HTTP stands for Hypertext Transfer Protocol and refers to a stateless protocol used to transfer data in an IP network. It was conceived to transfer web pages and data between a web server and a web browser. HTTP is also used by WebDAV, the file transfer protocol, and REST (**rest**). In the OSI layered model, the protocol is assigned to the application layer (Layer 7). The protocol was developed in 1989, and the HTTP/1.0 version was specified in RFC 1945 in 1996. HTTP/1.1 followed in 1999 (RFC 2616) and HTTP/2 in 2015 (RFC 7540).

[12] `https://io500.org`
[13] `https://wiki.lustre.org/POSIX_Compliance_Testing`
[14] `https://docs.ceph.com/en/latest/cephfs/posix/`

HTTP transmits information unencrypted in plain text, and HTTPS (Hypertext Transfer Protocol Secure) enables secure connections with authentication and end-to-end encryption. As a rule, the web servers authenticate themselves to the web browser with a certificate. The port used is port 443 instead of port 80.

HTTP also offers the option of user authentication. A web server sends the status code 401 and a WWW-Authenticate header field to inform the client that an authentication is required. The simplest form of authentication is basic authentication; the client sends an HTTP request containing an Authorization header containing the base64 encoded username and password joined by a single colon. If the credentials are correct, the server delivers the requested file.

HTTP also offers compression to reduce the amount of data transferred; the server can compress its responses. Client and server negotiate the compression method to be used beforehand. Especially with text-based data such as HTML, CSS, or JavaScript, compression can save a lot of bandwidth. For already compressed data, such as images, videos, or audio files, on the other hand, recompression makes less sense.

HTTP 1.1 defined in rfc2616 the methods used by HTTP, like GET to retrieve information from the server and POST to deliver the requested information.

2.5.1 HTTP2

HTTP1.1 clients use various connections to improve concurrency and decrease latency; HTTP/1.x does not compress request and response headers, generating redundant network traffic. Many techniques are used to overcome those limitations, like HTTP Pipelining or domain sharding, which introduced their own complexity level and increased network resources usage.

"HTTP2 enables a more efficient use of network resources and a reduced perception of latency by introducing header field compression and allowing multiple concurrent exchanges on the same connection... Specifically, it allows interleaving of request and response messages on the same connection and uses an efficient coding for HTTP header fields. It also allows prioritization of requests, letting more important requests complete more quickly, further improving performance."

"The resulting protocol is more friendly to the network, because fewer TCP connections can be used in comparison to HTTP1.x. This means less competition with other flows, and longer-lived connections, which in turn leads to better utilization of available network capacity. Finally, HTTP2 also enables more efficient processing of messages through use of binary message framing."(Hypertext Transfer Protocol version 2, Draft 17)

HTTP2 focuses on three qualities seldom associated with a single transport protocol without any additional network technologies: Simplicity, High Performance, and Robustness. Techniques such as compression, multiplexing, request prioritization, or server push enable HTTP2 lower the latency of processing the HTTP requests while maintaining interoperability and compatibility with HTTP 1.1.

- Multiplexing: As seen in fig. 2.9, several requests are sent over the same TCP connection, whereas responses can be received in random order without the need to wait for a slow response, that's blocking others in a way similar to out-of-order instruction execution in modern CPUs.

- Compression: HTTP header size is drastically reduced, minimizing the overhead.

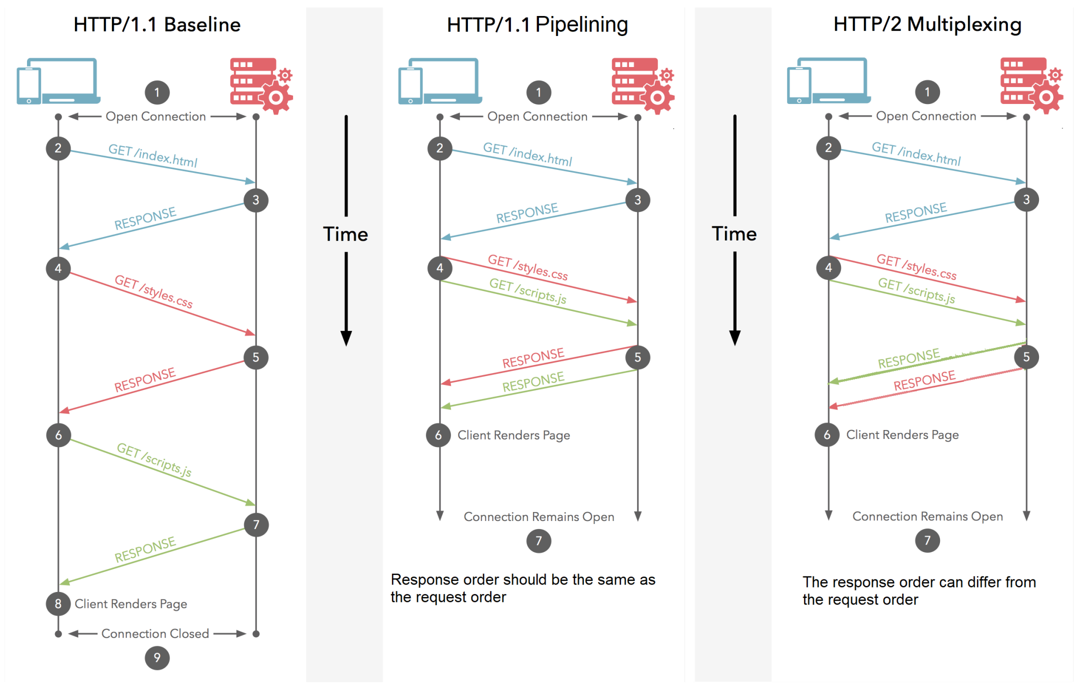


FIGURE 2.9: HTTP1 Vs HTTP2

- Server push: The server can send more content than initially requested by clients, diminishing the need for users to continually request information needed to fully load the web page in the web browser. As such, the server push capability with HTTP/2 permits the servers to respond with the full content of a page that is not already in the browser's cache.

2.5.2 HTTP3

Like its predecessor, HTTP/2 relies on the Transmission Control Protocol (TCP). In simple terms, the protocol ensures that data sent in a particular order also arrives at the receiving device in the same way. However, if a packet from this sequence is lost, the entire TCP connection is interrupted until the missing link arrives. Suppose the connection medium exhibits a high loss rate. In that case, this can lead to HTTP1.1 ending up being faster than its successor - because HTTP1.1 establishes up to six TCP connections, over which the lost packets would then be distributed. The individual connections are then less affected than the one TCP connection for HTTP/2, over which multiple data streams are transmitted.

Google has therefore been working on an alternative under the name QUIC since 2012. The technology was officially introduced a year later and was initially used for internal communication between Google servers and later also used by Facebook. QUIC no longer relies on the connection-oriented TCP but on the connectionless User Datagram Protocol (UDP). UDP does not require explicit connections to be established during data transport and does not require acknowledgment of incoming data. Any error corrections that may be necessary are carried out at the QUIC level, and, in the end, the number of packets is reduced, resulting in a significant acceleration. The Internet Engineering Taskforce (IETF), responsible for standardizing Internet transmission protocols, later agreed to use QUIC as the basis for HTTP/3. Individual data streams are handled separately in HTTP/3. Therefore, if a packet is lost en route, this no longer affects all data streams, as with a TCP transmission.

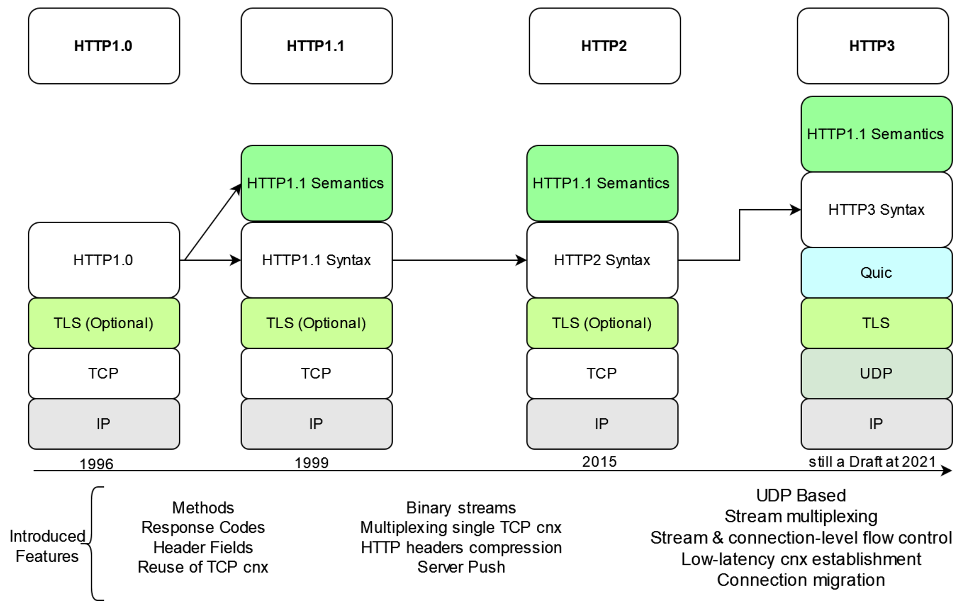


FIGURE 2.10: HTTP Protocol Evolution

Instead, with HTTP/3, only the stream that is actually affected has to wait until the missing packet is delivered. HTTP/3 also reduces the packet roundtrip time, and this is the time it takes for a data packet to arrive at its destination in addition to the time it would take to acknowledge receipt of the data.

In chapter 4, a benchmark is performed to test the different HTTP protocols, and a comparison of their performance is presented.

2.6 Performance Testing

The purpose of Performance testing is to evaluate the system or software in terms of different factors such as performance, responsiveness, scalability, stability, reliability, speed, or resource utilization. Effectively, the system is tested under various loads and network conditions while accurately recording and evaluating its response time.

Various test scenarios are used for different attributes of the system and summarized as a report. The main thing to remember here is that some factors are interdependent. Further tests must therefore not be considered individually in performance testing, as it is ultimately a question of the system's overall performance for the end customer.

Ultimately, the correct evaluation of the results also requires that the tester knows what the clients finally expect from the system. There are some key performance indicators to be considered when evaluating various tests. They provide important indicators of how the system is performing and enable the comparison to other systems.

These key performance indicators include the number of virtual users, calls, errors per second, response time, resource utilizations, waiting times, execution demands (for CPU cycles or seconds) and throughput. By evaluating this data and its correlations, we can better identify errors and bottlenecks that limit the system's performance. The system's scalability is determined by assessing the variation in response time/resources as the load on the system changes.

Performance testing is generally divided into two subtypes, load and stress testing. Load testing increases the load volume, so this performance test shows how a system behaves under a large number of regular, expected requests. Load testing helps us determine the system's reliability by assessing the key performance indicators mentioned above under prolonged periods of high load. In contrast, stress testing is performed to find out how software behaves under maximum demands. Load Testing tests the extent while Stress Testing tests the peak of the requirements. Stress Testing is essential to test the system's performance, response time, stability, and recovery and close eventual security gaps.

2.7 Convergence Scenarios

In recent years, we have heard a lot about the convergence of Cloud and HPC. It has been advertised as the next potential trend in the HPC industry and the inevitable way to achieve more affordable and faster HPC systems. However, many aspects seem to fall under the broad term "Cloud and HPC Convergence," so before diving into the different types of convergence, let's define the term convergence. The best-found definition related to our context is brought by (Wikipedia contributors, 2021), stating that the *"technological convergence is the tendency for technologies that were originally unrelated to become more closely integrated and even unified as they develop and advance".*

From this definition, we can see that the primary purpose of convergence is the possibility for different technological systems to emerge toward performing similar tasks. The resulting converged system can seamlessly accomplish any types of tasks that were previously completed on only one of the past non-converged systems.

The following scenarios claim a certain extent of HPC Cloud convergence.

2.7.1 Cloud Bursting

Cloud bursting refers to the ability to dynamically move workloads from an HPC Data Center to the cloud to meet peak demands or free up local resources, thus maintaining on-premises workloads at targeted levels (Sabin et al., 2016).

Some Industry trends claim that hybrid environments can offer the best results where heavy or sensitive workloads can operate on on-premise resources, and peak demand can be moved to remote resources following a pay-as-you-go model.

An example of cloud bursting is the solution offered by the company Adaptive Computing [15] for clients running its Moab job scheduler. This approach binds Moab with the NODUS Platform in order to operate compute nodes in the cloud and offers HPC users the capability to seamlessly manage and configure cloud-based nodes that will handle the extra workload and incorporate those nodes into the customer's on-premises infrastructure.

One of the advantages of this approach is the possibility to burst to different cloud providers like AWS[16], Google, Azure, etc. Cloud resources are automatically de-provisioned from the cloud provider after usage.

Cloud Bursting might seem like an ideal approach to achieve huge savings while offering endless resources to the local HPC users; however, several aspects should be carefully evaluated to avoid any pitfalls:

- Data Synchronization:
  The amount of data handled by the HPC workload decides whether it can

[15] `https://adaptivecomputing.com/adaptive-computing-makes-hpc-cloud-strategies-more-accessible-with-the-moa`

[16] Amazon Web Services

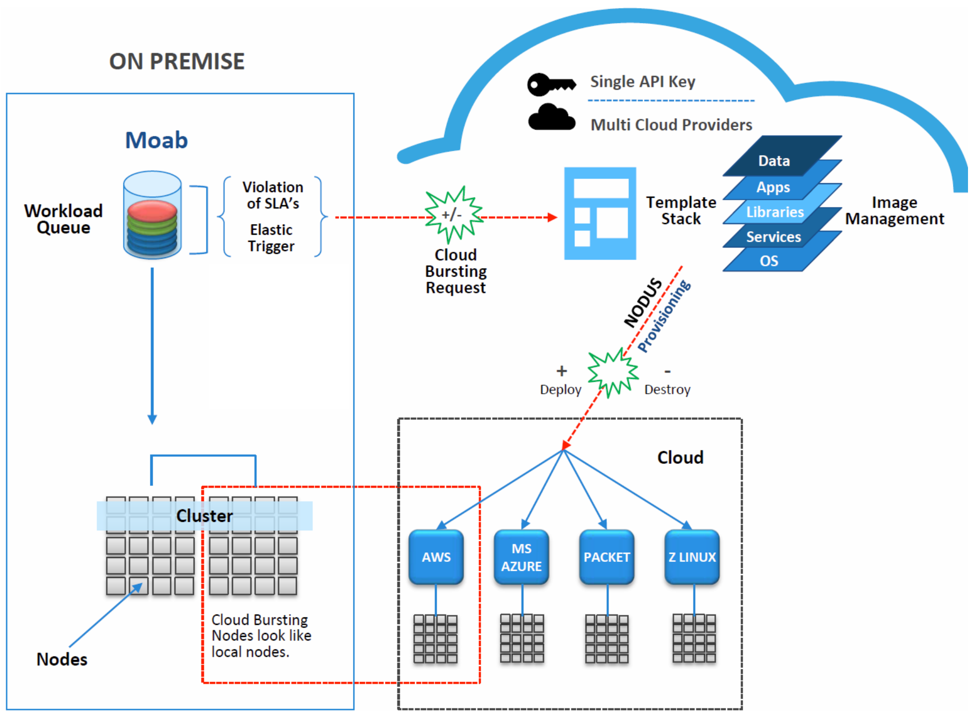


FIGURE 2.11: Moab Cloud Bursting

burst into the cloud. Simulations jobs, for example, require a small amount of data to start but may produce terra or even petabytes of data. Other jobs like weather prediction require hundreds of gigabytes of data before they can begin. Using the best internet connection might not suffice to synchronize this amount of data between the data center and the public cloud.

- Traffic Costs:
  As already discussed in chapter 1, we should not neglect the cost of the cloud egress traffic. Let's assume we can overcome the data synchronization challenge and send data and receive results seamlessly; however, if the applications produce lots of data that we should pull back to the data center, then the bill will likely increase drastically.

- Possible Code Change:
  To take full advantage of the cloud, it might be necessary to change the application code to interact with the new infrastructure. The complexity of this code change increases based on the constraints set by the cloud provider and the extent to which the offered services diverge from standards. The application should also be aware of the runtime environment, whether on-premises or in the cloud, and operate accordingly.

- Workflow Dependencies:
  This might happen when some of the on-premises workflows have specific software or hardware dependencies that are hidden or just are not available in the cloud. This is designated by the complexity of the application and its dependency and integration with other applications, components, and systems internal to the data center.

Cloud bursting can offer great flexibility, scalability, and elasticity; nevertheless, it is mainly suitable for stateless, non-critical applications that handle non-sensitive

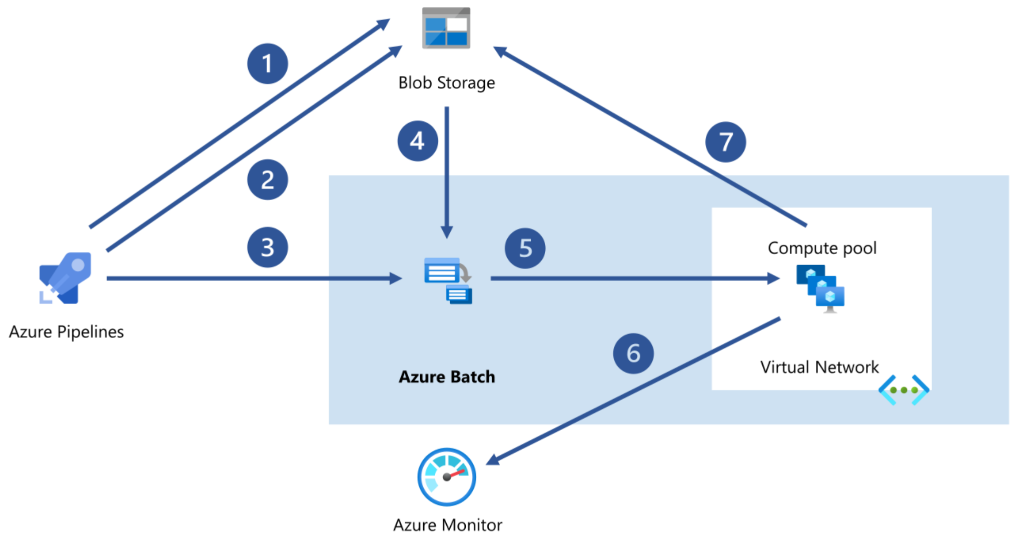


FIGURE 2.12: HPC Cloud using cloud-native technologies - as offered by Microsoft Azure

information. Since this only covers a minimal subset of HPC workloads, let's see what other convergence scenarios offer.

2.7.2 HPC Cloud

The term *HPC Cloud* - a.k.a. HPC in the Cloud or HPC as a Service - was introduced in chapter 1, it refers to the high-end hardware platforms and software environments offered by some Cloud providers to run HPC applications.

The key concept is the use of cloud resources to run HPC applications (Netto et al., 2018).

In recent years, several scientific applications and business analytics services have been executed on HPC Clouds. As such, current research efforts seek to understand the cost-benefit of moving resource-intensive applications from on-premise environments to public cloud platforms.

We can achieve the simplest type of HPC Cloud by just spinning up a couple of virtuals servers in the Cloud then installing and configuring an MPI(Message Passing Interface) library like, for example, OpenMPI (Graham, Woodall, and Squyres, 2005) in cluster mode.

The major cloud providers offer more realistic examples. We can differentiate between two types of HPC Cloud offerings:

HPC in the Cloud using 100% cloud-native solutions

A direct example is the HPC solution built on the Azure managed service[17]: Azure Batch and started by an Azure Pipelines job.

The solution shown in fig. 2.12 involves the following steps:

- An Azure Pipeline is launched to compile the project's code an generate an executable stored in Azure Storage
- The pipeline job loads processing data to the storage.

[17] https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/hpc-cluster

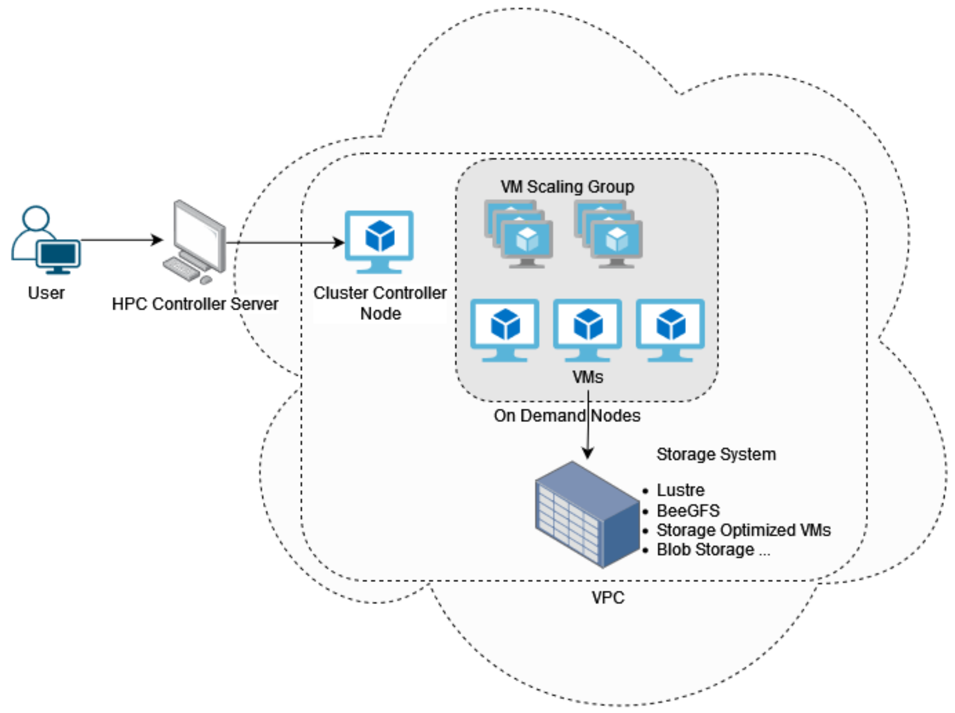


FIGURE 2.13: HPC Cloud using HPC technologies

- Azure Pipelines triggers the Azure Batch service to initiate its processing job.
- The Azure Batch copies the program executables and the input data from storage and attribute it to a compute nodes pool.
- The Batch service performs the job while managing the pool by retrying or reassigning tasks as nodes complete their work.
- The pool performance metrics (CPU, Memory, Disk I/O) and log files are collected by the Azure Monitor service.
- Once the compute nodes complete the tasks, the program data is delivered back to the Azure Storage.

HPC Cloud using HPC technologies

In this case, the cloud provider offers HPC Technologies like parallel filesystems, RDMA over Infiniband, and VMs running on hosts equipped with special hardware, i.e., GPU support. A typical use case is the Computational Fluid Dynamics (CFD) simulation which requires notable compute time and specific hardware. This increase in cluster usage can eventually lead to problems with spare capacity and prolonged queue times. Since adding physical hardware might be expensive or may not arrive within the given deadline, some clients consider using this type of HPC Cloud to conduct their research.

As seen in fig. 2.13, the cloud provider offers the hardware to run the CFD jobs on both GPU and CPU virtual machines. RDMA (Remote Direct Memory Access) enabled VM sizes with FDR InfiniBand-based networking allows low latency

MPI transfer. An enterprise-scale clustered file system is offered to ensure the best throughput for I/O operations.

The cloud provider might also offer specific tools to provision clusters and orchestrate data in both hybrid and cloud scenarios, in order to streamline the creation, administration, and optimization of HPC clusters. It is also possible to automatically launch on-demand compute nodes by monitoring the number of pending jobs.

It is inevitable to use provisioning tools to set up the significant number of nodes constituting a typical HPC system. Many tools (Scott, 2001; Bruno et al., 2004; Schulz et al., 2016) have been developed to accomplish this. The OpenHPC project [18], for example, seeks to aggregate several components needed to deploy and operate Linux HPC clusters, including resource management, provisioning tools, I/O clients, development tools, and a mixture of scientific libraries.

We can also use most of these tools to seamlessly provision VMs in the cloud (Trangoni and Cabral, 2012); furthermore, there is an abundance of tutorials explaining how to set up an HPC Cloud. Since many clients do not have the time or qualified human resources to endeavor in the cloud, several small companies are now addressing this market segment by offering their expertise in setting up HPC Cloud environments.

2.7.3 HPC Grid

Grid Computing is defined as a network of uniform or different computers working together *over a long distance* to accomplish a task that would rather be challenging for an individual machine. Grid computing is suitable for a subset of HPC applications called high-throughput computing (HTC); these applications are generally loosely coupled, where communication between processors is limited or even non-existent. As such, HTC applications demand large amounts of computing, while a high-speed interconnect isn't crucial. Typical case studies include protein folding, financial modeling, and earthquake simulation.

A grid computing software needs to be installed on each node to enable grid participation. The Berkley Open Infrastructure for Network Computing (BOINC) (Anderson, 2004) is an open-source grid middleware used extensively in scientific research to create and control the grid and manage processes and communication across the nodes.

Such open-source middlewares leverage "Volunteer Computing", which is the "use of consumer digital devices for high-throughput scientific computing" (Anderson, 2020), and enable them to support high-performance workloads; for example, the folding@home project (Pande et al., 2010) managed to achieve 1.1 exaFLOPS in March 2020. The main challenges facing HPC Grid were addressed by (Tanaka et al., 2013) and can be summarised as follows:

- Data retrieval and sharing over the internet.
- Network virtualization.
- Resources management and load balancing.
- Security.

[18] `http://www.openhpc.community`

2.7.4 Cloud Technology inside HPC

To extract the top performance from the underlying platform, and to face increasing demands for greater software flexibility, while achieving the correct pricing and contractual models, many efforts are spent on adopting cloud-native technologies like containers and object storage inside HPC data centers, which in turn will help to reach harmony between the two worlds. The benefits can be summarised as follow:

- **Portability**: Containers encapsulate software dependencies to move the application between machines/clouds (Docker, Kubernetes, etc.) without worrying about the re-installation or the interoperability of the required libraries. On the other hand, object storage is more flexible than traditional POSIX filesystems because it can be accessed from any client that supports HTTP (S3, Ceph ..)

- **Scalability**: Containers decompose large, monolithic HPC applications into more scalable microservices, not tied to a particular infrastructure. On the other hand, the sheer number of Object or Cloud storage providers claim Infinite scalability [19], which literally means there is no restriction on how large the system and the data size could grow. This claim will be quantified and analyzed in chapter 5.

- **Performance**: The containerization overhead is negligible. (Torrez, Priedhorsky, and Randles, 2020) found no meaningful performance differences when running a set of benchmark container solutions in comparison to running them directly on bare-metal; they only noticed a possible exception of modest variation in memory usage. They also point out that HPC users should feel free to containerize their applications without being worried about performance degradation, regardless of the container technology used. The overhead of cloud storage is explored in-depth in chapter 4 and chapter 5.

Containers in HPC

The most notable containerization solution explicitly tailored for HPC environments are:

- **Singularity**: This solution provides isolation of the different workloads while preventing privilege escalation. Singularity (Kurtzer, Sochat, and Bauer, 2017) offers native support for MPI, Infiniband, and GPUs. It supports storing images and running containers in HPC standard file systems due to its ability to distribute container images as a single SquashFS[20] file. Singularity focuses on security by encrypting the root file system and cryptographically signing the containers' images. Furthermore, the started containers are read-only by default, and all the write operations are only allowed using bind-mounted directories.

- **Charliecloud**: Another HPC-focused containerization solution is Charliecloud (Priedhorsky and Randles, 2017), which also natively supports MPI. Charliecloud differs from Singularity by focusing on the simplicity of architecture to run standard Docker containers without any privileged operations. It mainly aims to encapsulate dependencies with minimal overhead. It uses Docker, among others, as an image builder tool and then extracts all the contents of

[19] https://www.netapp.com/data-storage/storagegrid/what-is-object-storage/
[20] https://www.kernel.org/doc/html/latest/filesystems/squashfs.html

that image into an unpacked file tree format. As such, the images are stored as a file tree, in contrast to Docker, which uses compressed layers, or Singularity, which uses a single file.

Other cloud technologies have also found their way into HPC, like, for example, Hadoop[21] the Big Data analysis platform, used to enable users execute data analysis applications handling vast amounts of distributed data. Hadoop uses an open-source version of MapReduce that minimizes overhead in task spawning and data communication. Tools like Spark (Zaharia et al., 2010) have gained popularity within the Hadoop ecosystem by addressing typical data processing and analytics needs used in sciences, like for example, DNA sequencing and Bio-molecular simulations (Massie et al., 2013). These applications generate immense amounts of data that need to be analyzed to choose the next best set of simulation configurations. The results obtained in (Reyes-Ortiz, Oneto, and Anguita, 2015) from experiments with a particle physics data set show than native HPC protocols like MPI/OpenMP outperforms Spark in terms of processing speed and presents a more consistent performance. However, Spark offers better data management and data security, by addressing aspects like node failure and data replication. On the other hand, several frameworks emerged to accomplish interoperability and integration between Hadoop and HPC, MyHadoop (Krishnan, Tatineni, and Baru, 2011), JUMMP (Moody et al., 2013), MagPie [22] are just a few examples. SAGA-Hadoop[23] is also a tool used for the deployment of Hadoop and Spark on HPC resources. It allows executing an application written for YARN (e. g.MapReduce) or Spark (e.g., PySpark, DataFrame, and MLlib applications) on HPC resources. SAGA-Hadoop uses SAGA (Merzky, Weidner, and Jha, 2015) to spawn and manage Hadoop clusters within an environment controlled by an HPC scheduler, such as SLURM[24].

Cloud Storage for HPC Workloads

As seen earlier, a certain overhead is associated with the interoperability between Big Data and HPC; data must to be transmitted which leads to persisting files and re-reading them into Spark.
While many cloud vendors offer to move complete HPC workloads into the Cloud, this is limited by the massive demand of computing power alongside storage resources typically required by I/O intensive HPC applications. Eventually, the cost of storing and managing data produced by those applications determines where workloads should run. It is widely believed that HPC hardware and software protocols like MPI yield superior performance and lower resource consumption compared to the HTTP transfer protocol used by RESTful Web Services that are prominent in Cloud execution and Cloud storage. With the advent of enhanced versions of HTTP, it is time to reevaluate the effective usage of cloud-based storage in HPC and its ability to cope with various types of data-intensive workloads.
The need for a common storage interface is crucial to achieving a seamless convergence between HPC and Cloud. Cloud storage offers lower overhead and better performance by disregarding primarily unused features like file hierarchies. The usage of cloud storage instead of distributed file systems for big data analytics or within storage abstractions approaches like key-value stores, is gaining more consideration. (Matri et al., 2017) points out that *"cloud storage is a strong candidate for*

[21] https://hadoop.apache.org/
[22] https://github.com/LLNL/magpie
[23] https://pypi.org/project/SAGA-Hadoop/
[24] https://slurm.schedmd.com/overview.html

replacing traditional storage for both HPC and Big Data by demonstrating that the majority of the storage calls made by HPC or Big Data applications can be seamlessly mapped to cloud storage operations."

Another example of the use of Cloud storage is HSDS [25], is the Object Store-Based Data Service for Earth System Science: Due to the massive increase in the volume of data in the Earth Observing System Data and Information System (EOSDIS) archive, NASA decided to move it to the cloud. This move fosters the management and accessibility of NASA Earth observation data by bringing this massive of data "close to compute" in an efficient, simple, and scalable way. Researchers are now able to access considerable amounts of data using the same applications they use on their own machines without bothering about storage or computing constraints. The resulting Highly Scalable Data Service (HSDS) (HDF-Group, n.d.) is a REST-based service used for reading and writing HDF5 data. It supplies all the functionality that the HDF5 library usually offers but in a way that can use cloud-based storage (e.g., AWS S3).

2.8 Summary

In this chapter, the background work for the following chapters was laid down. The significant terms and notions were introduced, and the HPC relevant access technologies were presented and successfully compared. A brief history of the HTTP protocol and its evolution was given. The HPC and Cloud convergence scenarios were also revealed and explained. In the next chapter, we will explore in-depth these convergence scenarios by introducing a modeling approach to compare them.

[25] https://earthdata.nasa.gov/esds/competitive-programs/access/hsds

Chapter 3

Research Methodology and Design

This chapter lays the groundwork for the upcoming chapters by introducing the methodology needed to address the HPC and Cloud convergence in-depth. One of the main goals of this work, defined in chapter 1, is to quantify the level of HPC and Cloud convergence. Hence, we introduce a model-based development technique to quantify the level of convergence reached by each solution. The need for such a model was outlined in chapter 1; in this chapter, we compare the different solutions already presented in chapter 2 by offering abstract and domain-specific concepts to represent each implementation.
Section 3.1 outlines the research methodology for the rest of this work, section 3.2 introduces our assessment model consisting of three distinct components: section 3.3 starts by assessing the ***performance feasibility****, then section 3.4 illustrates the second component which is the* ***administrative overhead****, section 3.5 concentrates on the third component to estimate which is the* ***cost efficiency****. Using the assessment metrics defined for each component, we evaluate the various convergence scenarios and evaluate the achievable level of convergence. A real-world scenario is also presented in section 3.5.5, where the cost of running workloads and storing their results on-premises and in the cloud is carefully examined. Finally, in section 3.6 the findings are discussed and analyzed, and a complete comparison of the various convergence models is provided.*

3.1 Research Methodology

As per the definition seen in the previous chapter, the primary purpose of convergence is the possibility for different technological systems to emerge toward performing similar tasks. The resulting converged system can seamlessly and efficiently execute any types of workloads previously conducted on only one of the non-converged systems, i.e., in our case, both HPC and Cloud workloads.

The solutions presented in chapter 2 claim to achieve a certain amount of convergence between HPC and cloud; each solution is located and uses the infrastructure of one of the two platforms, sometimes both and even it may extend to use the compute power provided by volunteer computers as in the case of HPC Grid.

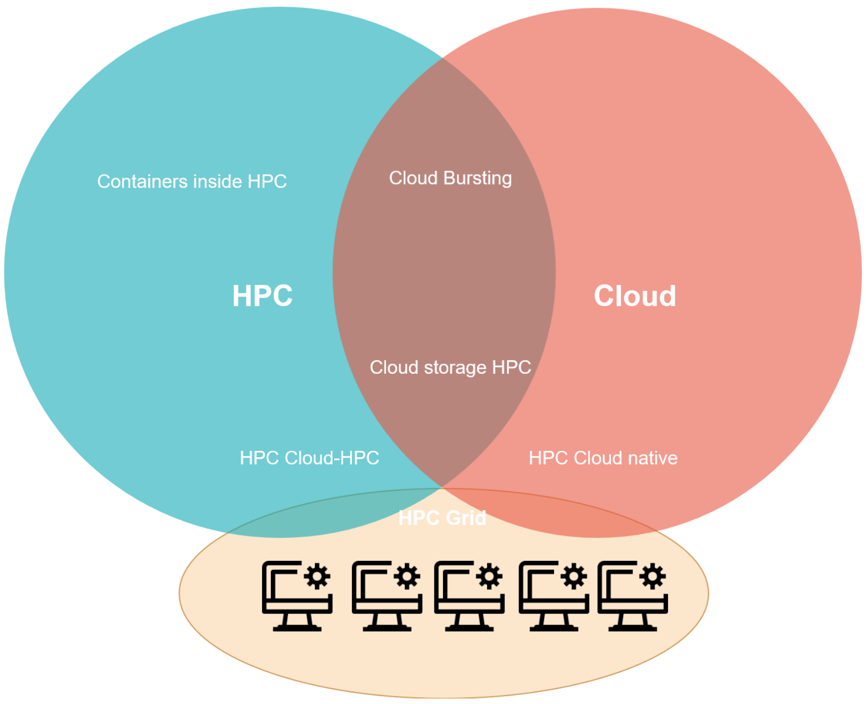


FIGURE 3.1: The different converged solutions with their positioning

Figure 3.1 summarizes the various scenarios presented in chapter 2 alongside their respective emplacement.

As stated in section 1.3.1 there is a need for an assessment model to quantify the level of HPC and Cloud convergence. This assessment model will help us answer some of the high-level questions posed in section 1.3.1 like the one related to the cost-efficiency of moving applications from on-premises to the public cloud. It will also help us examine the different solutions already presented in chapter 2 and compare them to each other.

A direct result of this comparison is the ability to determine the most promising solution, which we will cover in-depth in the upcoming chapters. Once identified, we need to determine the performance bottlenecks and scalability problems in this system for both HPC and cloud applications. For this purpose, we use well-known and established HPC and Cloud benchmarks. Furthermore, to answer the question posed in section 1.3.1, regarding the simultaneous use of HPC and cloud storage technologies and what workflows will benefit from such settings, we need to extend the benchmarks mentioned above to handle both types of workloads; hence the same benchmark should be able to reproduce HPC and Cloud workloads by simulating various types of I/O intensive applications using the access interfaces typical found in each world, i.e., POSIX or MPI-IO for HPC, and Cloud Storage like S3 for the cloud.

Another high-level question from section 1.3.1, regarding the overhead to expect when using cloud storage in an HPC environment, can be addressed just by using the extended benchmark inside HPC and comparing the results obtained of the runs against HPC and Cloud I/O interfaces.

After defining the possible causes of the overhead of using cloud storage inside HPC , it will be possible to propose different workarounds to overcome these significant bottlenecks and scalability issues. Implementing these workarounds and re-running the same benchmarks will help us determine the efficiency of the optimized solution.

3.2 Convergence Assessment Model

One of the main goals of this work, already seen in section 1.3.1, is assessing the level of convergence between HPC and Cloud. To accomplish this, we introduce an assessment methodology to measure the extent of this convergence achieved by a converged system.

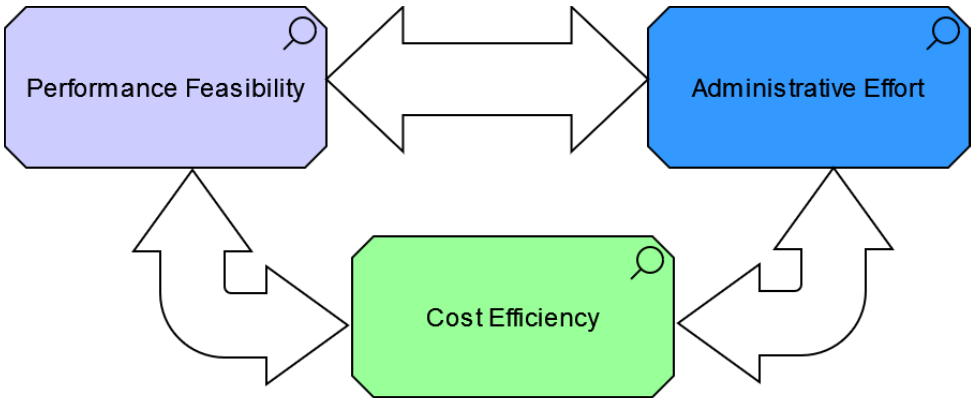


FIGURE 3.2: HPC Cloud Convergence Assessment Model

Figure 3.2 represents a high-level overview of our assessment model, which we will elaborate on in the upcoming sections.
Our model comprises three related but distinct components: It starts by assessing the **performance feasibility** of the converged model to determine if it can meet the performance and scalability constraints imposed by typical HPC and Cloud applications.

The second component is the **administrative overhead** which includes the human resources used to manage and maintain a converged environment.

The third component to estimate is the **cost efficiency**, which is a significant cause affecting the choice between running workloads on-premises or on the cloud, alongside the type and the hardware and software requirements of the application. We also check to what extent investing in on-premises resources is meaningful.

3.3 Performance Feasibility

As seen in section 2.7, an HPC Cloud converged system is capable of seamlessly running any types of workloads that were previously completed on only one of the non-converged systems, i.e., both HPC and Cloud workloads.

Figure 3.3 shows the different aspects that might generate an inevitable overhead, limiting the overall performance and the performance feasibility of a converged environment. Our primary interest, in this case, is not just the technical feasibility of a particular solution but also the performance that each element of this solution can achieve. This coarse-grained model also depicts how the components relate to each other:

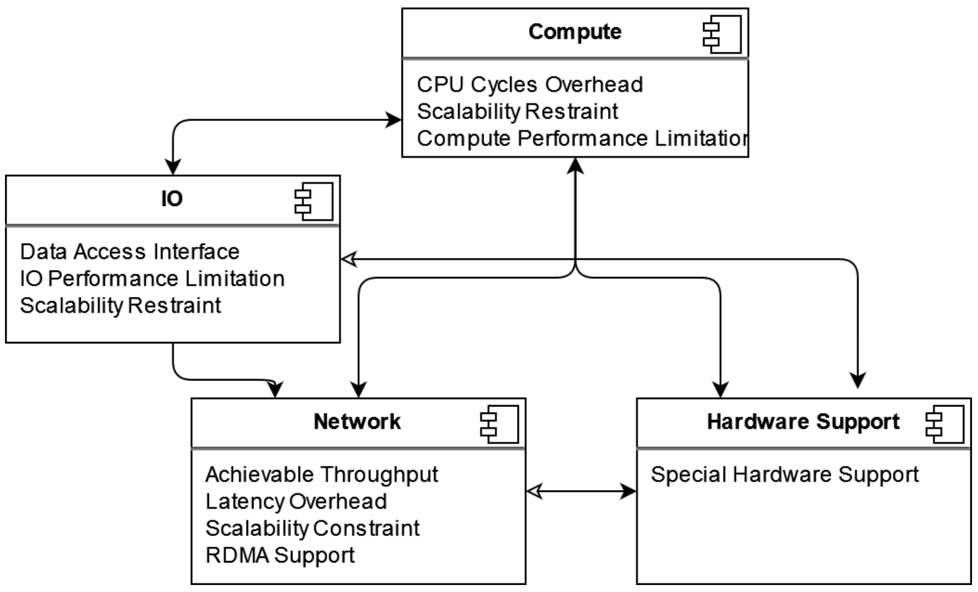


FIGURE 3.3: Performance Feasibility Model

- **Compute**: The computing overhead refers simply to the CPU cycle overhead added by the infrastructure itself. For example, running an application on virtual machines implies a particular overhead compared to running the application directly on bare-metal due to the hypervisor emulating the underlying hardware. Containers can be quite helpful since they offer some measure of isolation without the performance overhead usually associated with virtualization. They are smaller than a VM and require much less time to start, allowing more containers to run on the same host. However, both virtualization and containerization need a specific management software which also brings some compute overhead to operate and control the virtualized or the containerized platform (Felter et al., 2015). The total compute overhead can be quantified by the number of CPU cycles used to run the emulated environment. We will use the performance metric "CPU_CLK_UNHALTED.CORE" to represent core cycles when the core is not halted or just CUC[1] for short. The overhead ratio can be expressed using Equation (3.1) :

$$Compute\_Overhead\_Ratio = \frac{CUC_{converged\_system}}{CUC_{bare\_metal}} \tag{3.1}$$

  Furthermore, a scalable system should be able to use a notable amount of resources while still serving its purpose, as such **scalability** plays a tremendous role here: supposing the application is highly scalable on bare-metal and can be run on x nodes simultaneously; however, if in the converged environment, the maximum nodes count is limited to y < x, which might be due to several limitations introduced by the infrastructure management platform such as Kubernetes[2], the application itself (Abraham et al., 2015) or the hardware resources (Chen et al., 2020). The application performance, in this case, is limited to a maximum valued estimated by $\frac{y}{x}$ in comparison to running on bare-metal.

  Another point to consider here is the **compute performance** achieved by the converged platform, expressed using a standard rate indicating the number

[1]CPU_CLK_UNHALTED_CORE
[2]`https://kubernetes.io/docs/setup/best-practices/cluster-large/`

of floating-point arithmetic calculations systems can perform on a per-second basis or for short FLOPS.
We will use the term $\frac{CP_{converged\_system}}{CP_{bare\_metal}} \leq 1$ to represent the ratio of compute performance achieved when moving the workloads to the converged system.

- **Network/communication**: The achievable throughput and the latency are the metrics to consider in order to estimate the network overhead. For example, a scientific application, usually executed inside an HPC data center equipped with a high-end InfiniBand network, will suffer performance loss if run on a cloud environment, with limited bandwidth due to the networking technology in use or the usage of some throttling, auditing, or accounting mechanism to charge the users or achieve a large and secure multitenancy. Scalability also plays a role here, and it can be quantified by the capacity of the routing and switching backplane in every platform. RDMA (Remote Direct Memory Access) support is also considered: The zero-copy networking enables the network adapter to transfer data directly from one application memory to another, without copying data between application memory and the data buffers of the operating system. Furthermore, if parts of the converged system are geographically dispersed with relatively slow interconnect, it will lead to a noticeable network overhead for tightly coupled HPC applications.

  To quantify the change in the network throughput, we can calculate the throughput ratio between the converged system and an optimally highly networked environment, presumably HPC, using Equation (3.2) :

$$Throughput\_Ratio = \frac{Throughput_{converged\_system}}{Throughput_{HPC}} \tag{3.2}$$

- **Hardware support**: Cloud and HPC applications have different hardware requirements: many HPC applications nowadays require the use of high-end hardware, the support for this special hardware should be available in the converged system. For example, GPU support plays a meaningful role: According to the latest HPC User Site Census data, out of the 50 most popular application packages considered by HPC users, 34 offer GPU support[3]. Furthermore, the use of the architecture's highly parallel features should not be reduced when using the converged environment.

- **I/O**: The I/O overhead addresses the component involved in the data transmission and transformation. We consider the following:

  - **Data Access Interface**: The various workloads can use different I/O interfaces, like POSIX, MPI-IO, or a specific high-level I/O library for the direct object store access. We should carefully evaluate the overhead of the data access technology used in each case for each of these interfaces.
  - **I/O Performance Variation**: includes different metrics like throughput and latency; it can be estimated using a benchmark capable of running HPC and Cloud workloads on the converged environment using the different I/O interfaces mentioned above.
  - **Scalability Constraints**: refers to the capacity of the storage system to accommodate several clients concurrently without leading to degradation in the performance or a higher error rate. Depending on the supported

[3]https://www.nvidia.de/content/intersect-360-HPC-application-support.pdf

number of nodes, we can distinguish between a low scalable system, only capable of supporting up to five nodes without suffering any performance or resilience loss; if the number of supported nodes is between 5 and 100, the storage system is moderately scalable. If it supports more than 100 nodes, we refer to it as a scalable system, and if more than 1000 nodes are supported, we can safely declare it a highly scalable system.

Note that any network overhead due to shared storage is ignored for this element since it is examined in the network aspect.

To implement a converged system, we need to identify the overhead associated with each component of the defined model. We will use the heatmap technique to better visualize this overhead: the magnitude of the overhead introduced by each aspect will be encoded, where red represents the highest overhead and green is the lowest. For this purpose, we will use the color palette illustrated in fig. 3.4.

We will now apply this model to the different convergence scenarios described in the previous chapter.

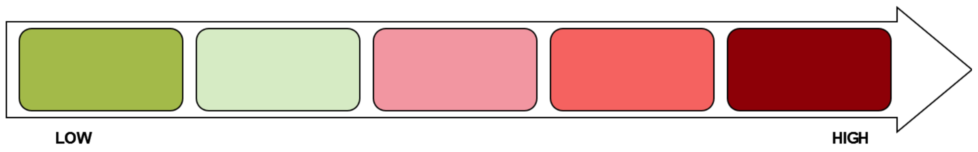


FIGURE 3.4: Color Palette used to display the overhead

3.3.1 HPC Cloud using 100% Cloud-native Solutions

As discussed in section 2.7.2, this scenario involves the use of cloud technologies inside a cloud environment to accommodate both Cloud and HPC workloads.
The compute overhead is the one introduced by virtualization; the VMs need to be provisioned and require a specific time to start compared to the bare-metal nodes usually provided in HPC.
Scalability should not be an issue, and, theoretically, it is only limited by the application itself.
When using the standard network of the cloud provider (Ethernet), the common HPC networking advantages are not available in this situation. (Expósito et al., 2013) assessed the performance of communications on the Amazon EC2 CC platform and found that the communication start-up latency is relatively high compared to bare-metal, i.e., around 5x, and the throughput is around half the expected value indicating a poor network virtualization support. Another research (Persico et al., 2016) on this subject found that the traffic management policies enforced by AWS and Azure may severely impact both the performance perceived by the customers and the measurement results.
Moreover, cloud storage is used extensively in this scenario; although most cloud storage solutions presented by the major cloud providers promise a high degree of scalability, the high-end library used to access this storage should be carefully evaluated since HPC applications do not usually support cloud storage access, requiring as such to emulate or to develop a new access interface.

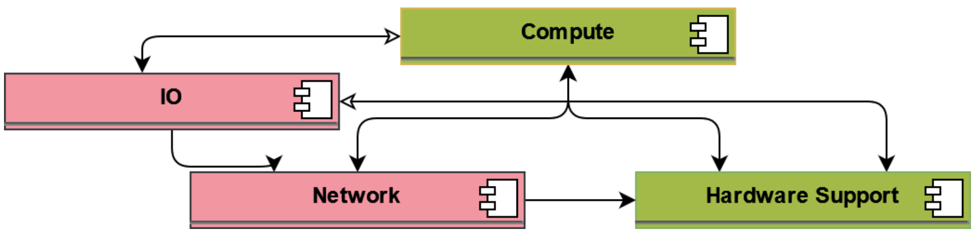


FIGURE 3.5: Performance Feasibility Model of the HPC Cloud using 100% Cloud-native Solutions

Therefore, fig. 3.5 depicts the low overhead for the compute and hardware support elements, whereas the network and the I/O parts suffer a higher overhead.

3.3.2 Cloud Bursting

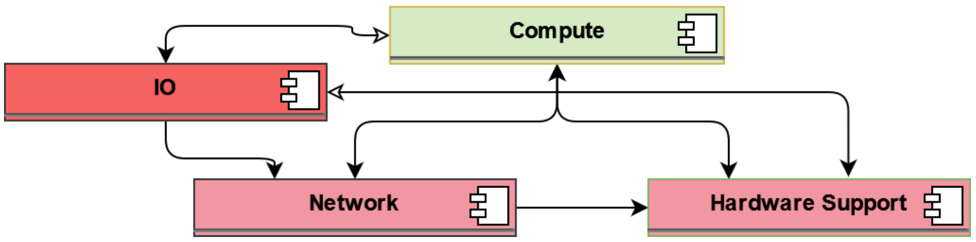


FIGURE 3.6: Performance Feasibility Model of Cloud Bursting

Cloud bursting refers to the ability to dynamically move workloads from an HPC Data Center to the Cloud to meet high demands or free up local resources, thus maintaining on-premises workloads at targeted levels.

Most issues of Cloud bursting were outlined in section 2.7.1, the different overhead points are now visualized in fig. 3.6, we can see that the compute overhead is acceptable, to a certain extent, it includes the virtualization overhead with the time needed to provision and start the virtual machines. However, the network and I/O components suffer at most. Even with the best internet connection available, this network channel will induce significant data-out charges and add latency to the application. Furthermore, data synchronization will strain the network and storage resources both on-premises and in the Cloud. Hardware support should also be provided in both environments if the application needs it.

3.3.3 HPC Cloud using HPC Technology

A more feasible and potentially convergent solution is the HPC Cloud using HPC technologies described in section 2.7.2. In this case, the cloud provider offers HPC Technologies like parallel filesystems, RDMA over Infiniband, and VMs running on hosts equipped with special hardware, for example, GPU support. Bare-metal cloud, consisting of fully dedicated servers within a cloud environment, also falls within this category.

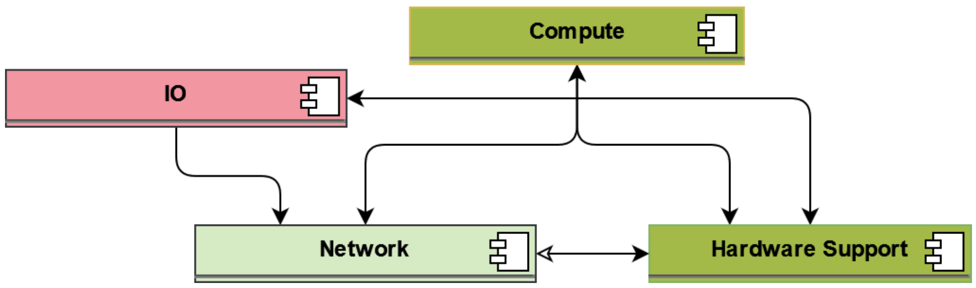


FIGURE 3.7: Performance Feasibility Model of the HPC Cloud using HPC Technology

Scientific workloads can be seamlessly run in this environment without any code transformation since the cloud provider offers an identical HPC environment. As such, hardware and network support should also be similar to what the HPC users know. One potential overhead can be the storage since parallel filesystems are emulated on top of a virtualized environment to provide the I/O storage interface. For example, running a SAS Grid Manager cluster on AWS with Intel Cloud edition for Lustre[4], the throughput obtained is still significantly lower than on-premises Lustre cluster.

Furthermore, Intel points out in the mentioned report that Lustre is ideal for running I/O-demanding workloads during the compute stage and not recommended for long-term storing data. They propose using the Cloud provider storage service, such as S3 on Amazon, for long-term data retention.

On the other hand, supporting InfiniBand in the cloud requires the use of particular types of virtual machines; for example, in AWS, the EC2 VM should have an Elastic Fabric Adapter (EFA). The team behind OpenFOAM[5], a software for computational fluid dynamics (CFD), found in their EFA benchmarks[6], that the scalability is to some extent low.

3.3.4 HPC Grid

As outlined in section 2.7.3, Grid Computing represents a network of uniform or different computers working together *over a long distance* to accomplish a task that would rather be challenging for an individual machine.
It is only suitable for a minimal subset of scientific applications, (Parashar et al., 2013) list the most common ones. The nodes exchange little or no data since most of them have limited network connectivity. The application code should ensure redundancy and robust failure recovery since it's highly likely that many compute nodes will disconnect or fail, increasing the compute overhead. Network and I/O performance are far from what is expected inside HPC due to the geographical dispersion of the nodes.

[4]https://insidebigdata.com/2017/01/26/hpc-storage-performance-in-the-cloud/
[5]https://openfoam.org/
[6]https://cfd.direct/cloud/openfoam-hpc-aws-efa/

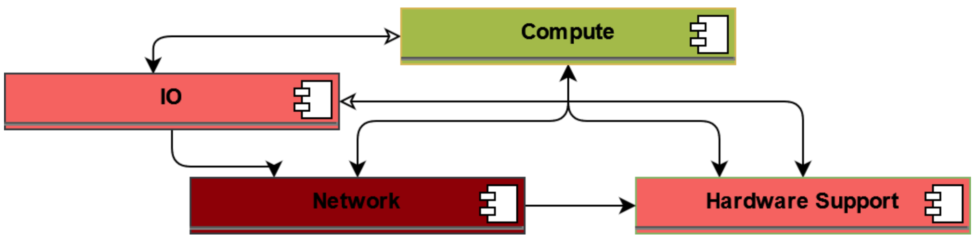


FIGURE 3.8: Performance Feasibility Model of the HPC Grid

3.3.5 Containers in HPC

The use of containers in HPC, as shown in section 2.7.4, brings many benefits like encapsulation, portability, and reproducibility. Security and prevention of root escalation privileges have also been addressed, and it is available in many HPC-specific container solutions or using a container orchestration solution like Kubernetes. By just importing a container and running it on the target platform, the development time elapses on the developer computer and not directly on supercomputers. As such, the cluster resources are preserved for production workloads and not for testing.

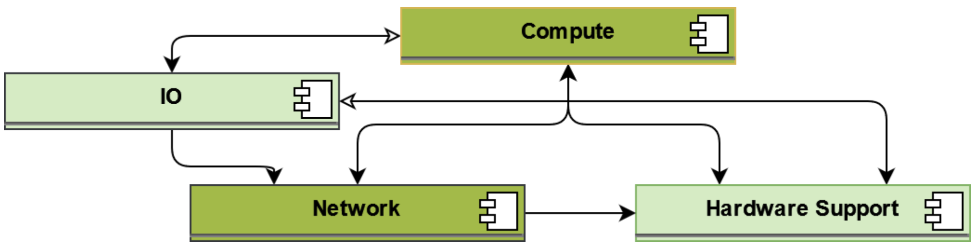


FIGURE 3.9: Performance Feasibility Model of the use of Containers in HPC

Although (Torrez, Priedhorsky, and Randles, 2020) - all members of the Charliecloud team - found no meaningful performance differences when running a set of benchmarks container solutions in comparison to running them directly on bare-metal, (Abraham et al., 2020) found a slight overhead when comparing different HPC container solutions. The evaluation done by (Abraham et al., 2020) shows a particular startup time overhead for Docker and Podman, as well as network overhead at startup time for Singularity and Charliecloud. The I/O evaluations show that Charliecloud incurs a significant overhead on Lustre's MDS and OSS with increasing parallelism.

Furthermore, integrating the containers with other HPC tools like schedulers and shared filesystem should be carefully addressed, inducing a particular overhead. Container images are built to use the commonly available network stack, i.e., TCP/IP; HPC typical interconnects like Infiniband and RDMA are not supported out of the box.

3.3.6 Cloud Storage in HPC

As seen in section 2.7.4, using a simple adapter for logging storage calls of HPC and Big-Data applications, (Matri et al., 2017) observed that HPC applications running MPI-IO do not perform any call other than file operations, with file reads and writes constituting the majority of them. On the other hand, 98% of the storage calls made by Big-Data applications are file operations. As such, the vast majority of the storage calls made by HPC or Big-Data applications can be directly mapped to cloud storage operations.

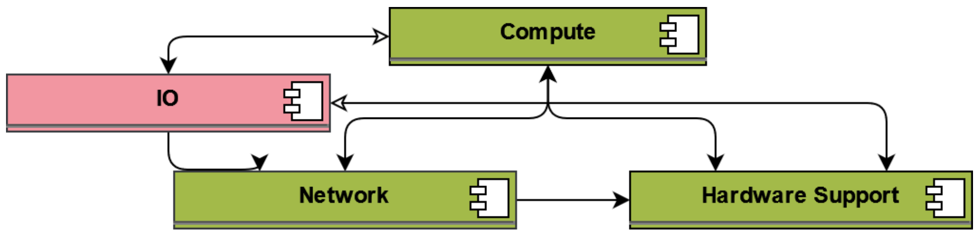


FIGURE 3.10: Performance Feasibility Model of the use of Cloud Storage inside HPC

Figure 3.10 shows the different components of our model. The color choice, in this case, is not definitive due to the lack of literature covering the use of cloud storage inside HPC. The compute overhead of using cloud storage boils down to analyzing the overhead of the communication protocol used by cloud storage, i.e., investigating the overhead of the REST protocol to reveal the potential for using cloud services for HPC storage.
Chapter 4 elaborates on this point, and chapter 5 focuses on the I/O overhead introduced by the use of such storage type.

3.4 Administrative Effort

A converged platform capable of running HPC and Cloud/BigData applications requires a competent IT staff who have a profound knowledge of both environments, a good understanding of the Pay-as-you-go model, and the ability to align IT and business strategies while keeping the environment **securely** and **reliably** running without incurring high costs.

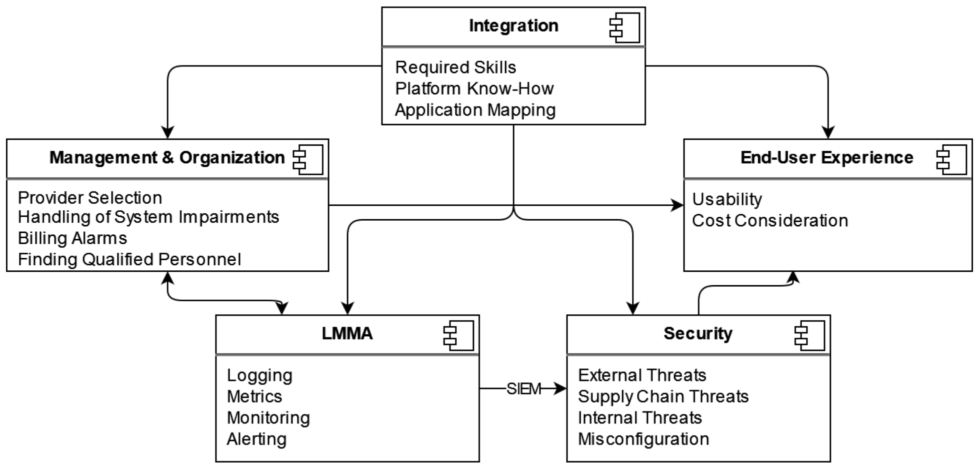


FIGURE 3.11: Administrative Effort visualized Model

The different aspects to consider are visualized in fig. 3.11 and can be summarized as follows:

- **Integration Overhead**
  The IT team should possess several **required skills** in order to get the applications running successfully. Here is a non-exhaustive list of the need-to-have skills:
  - Distributed computing
  - Virtualization
  - Containerization and experience in migrating legacy systems into cloud environments.
  - Web/REST API and Services
  - Design and implementation of Service-oriented architecture
  - Dealing with HPC and Cloud management tools
  - Solid understanding of the economics behind different HPC and cloud models.
  - Complete mapping of the applications and their dependencies
  - Full view of the existing integration options that the providers offer.

  Depending on the type of the converged solution, the learning curve can vary to acquire proficiency in those skills and achieve the needed **platform know-how**. For example, fig. 3.12 represents our vision of a cloud learning curve, figuring the milestones that should be reached to achieve a high level of proficiency and cloud platform know-how needed to operate and maintain cloud environments.

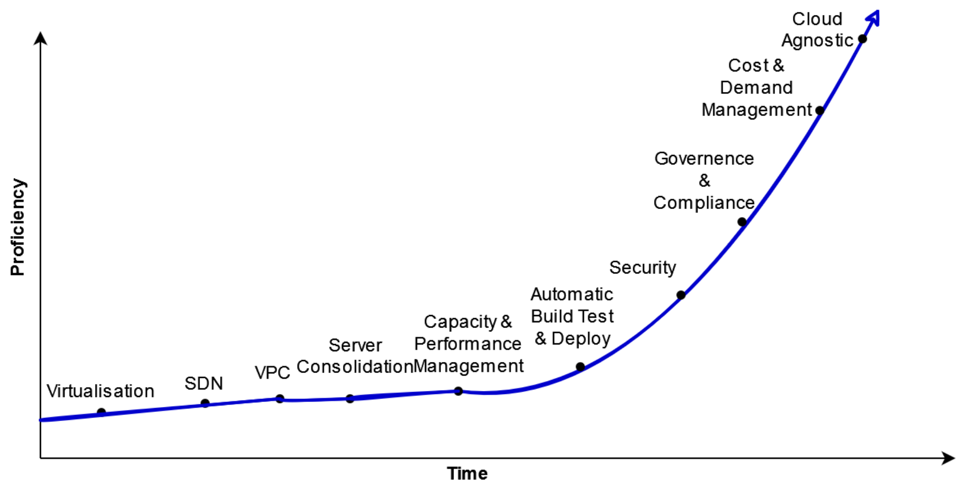


FIGURE 3.12: Cloud Learning Curve

Figure 3.12 displays the transition from general infrastructure knowledge to cloud platform mastery. This chart illustrates the challenges to expect when pursuing cloud capabilities from virtualization to software-defined networks (SDN) through Virtual Private Cloud (VPC), autoscaling, and automatic deployment.

Different software stacks are available on both cloud and HPC. Each environment has its own job scheduler, software dependencies, hardware drivers, and I/O interfaces; this is without mentioning the different software engineering workflows used by cloud and scientific developers, a.k a. the "software chasm"

(Kelly, 2007). Furthermore, the life cycle of supercomputers is five to seven times shorter than the life cycle of scientific applications that run on them (4 years versus 20 to 30 years) (Palyart et al., 2012). A scientific software stack usually outlives the hardware it was initially developed to work on.
The process of identifying and determining the interactions and interdependencies between application components and their underlying hardware infrastructure is expressed as **application mapping**. To ensure that the applications perform optimally in a converged environment, it's essential to discover and map the underlying dependencies.
A converged system should run both types of workload with minimal "code transformation," i.e., the application code should undergo a minimal or ideally no modification to run on the converged system.

- **Management and Organization Overhead**
  The management of a converged solution requires technological skills and the ability to design, deploy and administer the infrastructure in a way to align with business strategies. The tasks of the management team include the following:

  - The choice of the **right provider**, by specifying the institution's specific needs and aligning the cloud provider's platform and technologies with the organization's cloud objectives.
  - Avoiding any vendor lock-in, which might happen when the company adapts its processes or application code to accommodate certain vendor services or products, leading to a situation where the cost and complexity of switching to another vendor with better offers but is extremely high.
  - Depending on the type and complexity of the platform to operate and maintain and the number and kind of needed skills to achieve a seamless integration as described in the previous section, the management team should expect a particular overhead towards **finding the qualified personnel**.
  - Monitoring the incurred expenses and setting the respective **billing alarms** to avoid any cost trap and uncover hidden costs.
  - Implementing security measures to ensure high availability, backup, and abiding by regulatory conditions like data privacy and residency.
  - Defining and implementing the required procedures for avoiding and eventually handling **system impairments**, data loss, or even operator discontinuity.

- **End-user Experience**
  For the end-user, the **usage** of the system should be transparent; researchers should focus on their scientific work without being distracted by the setup and the security of the underlying infrastructure or having to use different configuration files for different environments. Most importantly, the **cost factor** should not be a significant concern that researchers should evaluate when running their workloads; for example, running an application in the cloud requires extra consideration to avoid massive expenses due to the **hidden costs** already described in section 1.2.1

- **Logging, Metrics, Monitoring, and Alerting (LMMA)**
  The LMMA[7] stack is a combination of tools used to ensure the availability of the running infrastructure. It should provide insight into how the applications and infrastructure are performing and help point out issues in load, networking, and other resources before it becomes a failure point. Metrics and Logs constitute the raw data needed by the Monitoring component to keep track of the system's performance, health, and availability. Alerting is triggered when performance or usage anomalies are detected. It is vital to define clear alert conditions that notify the qualified team or individual and have sufficient contextual information to help them decide how urgent an alert is.

- **Security**
  Security plays a critical role for every system, independent of its convergence level.**The greater the number of services accessible from the internet, the higher is the risk of being compromised.** The common source of risks are:

  - **External Threats**
    This is where the software vulnerabilities are exploited from a remote location to gain access to the system. Software vulnerabilities can stay hidden for a long time and might be exploited secretly before being made public, for example, Shellshock [8] found in the GNU Bash through 4.3, initially released on 08 June 1989, was only disclosed on 24 September 2014. Shellshock could allow an attacker to make Bash execute arbitrary commands and get unauthorized access to Internet-facing services, including web servers, that use Bash. Log4shell[9] is another example.
  - **Supply Chain Threats**
    Supply chain risk is when a software dependency or an element that makes up the system is compromised, including product components, services, or personnel that help supply the end product, third-party software, and vendors who implemented the system.

    A most recent example is the SolarWinds Hack in early 2020, where hackers secretly broke into Texas-based SolarWind's systems and added malicious code into the company's software management system. The hacked code was sent out as an update to the company's 33,000 customers[10]. The code created a backdoor to customers' information technology systems, which hackers then used to spy on them.
  - **Internal Threats**
    These describe staff members or users that abuse their access privileges. In a multitenant environment, special constraints should be applied to mitigate privilege escalation and audit user access.
  - **Misconfiguration**
    This happens, for example, when a specific staff member mistakenly exposes some service to the internet with a weak or no authentication at all, allowing a remote attacker to abuse this service easily.

Whether on-premises or in the cloud, the organization being the application owner, is responsible for protecting the applications, operating systems, and

[7]Logging, Metrics, Monitoring, and Alerting

[8]`https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271`

[9]`https://en.wikipedia.org/wiki/Log4Shell`

[10]`https://www.businessinsider.com/solarwinds-hack-explained-government-agencies-cyber-security-2020-12`

other assets running in either platform. This is well known when the organization manages its own infrastructure; however, it is less known that all the public cloud vendors point out that security in the cloud is a shared responsibility and that the client should secure the above-mentioned components, as outlined by fig. 3.13, courtesy of Microsoft[11] [12].

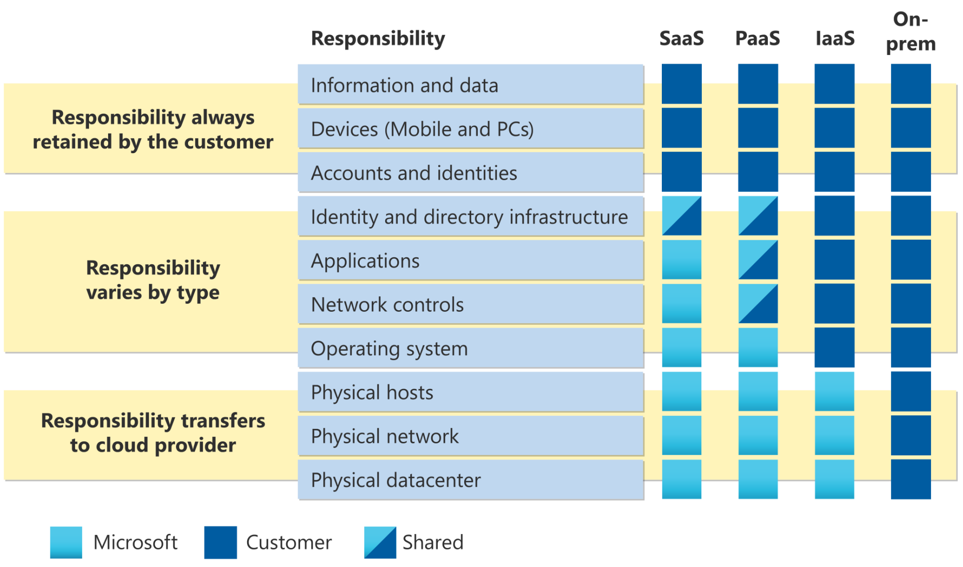


FIGURE 3.13: Shared responsibility in the cloud, as defined by Microsoft

The effort, time, and resources employed by the organization to mitigate against these risks represent the security overhead that this component model seeks to quantify. Depending on the level of the infrastructure exposure, the risks mitigation can be accomplished using the following procedures:

- Scanning of all software components for vulnerabilities or misconfigurations.
- Running workloads with least possible privileges.
- Network separation to limit the impact of a compromise, if it happens.
- Using firewalls to prohibit unneeded network connectivity and encrypt the traffic to enssure confidentiality.
- Using strong authentication and authorization mechanisms to restrict user and administrator access while narrowing the attack surface.
- Using log auditing which enables security administrators to monitor activity and defining alert rules to be informed of potentially malicious activity.
- Periodically reviewing the system settings and using vulnerability scans to discover potential risks and apply security patches where needed.

We will now see how our model applies to the different convergence scenarios described in the chapter 2; as seen earlier, the heatmap colors express the magnitude

[11] https://docs.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility

[12] SAAS: Software as a Service

of the overhead introduced by each of the aspects, where **red represents the highest overhead, green is the lowest**.

3.4.1 HPC Cloud using 100% Cloud-native Solutions

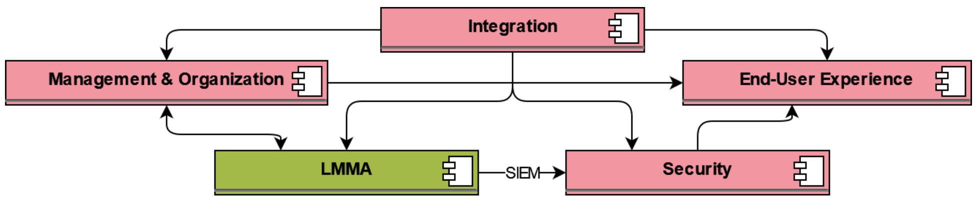


FIGURE 3.14: Administrative Effort of the HPC Cloud using 100% Cloud-native Solutions

This scenario is modeled in fig. 3.14.
The **integration** aspect is colored in red since this scenario requires a deep knowledge of the cloud environment and specific experience within the cloud toolset. Furthermore, the tools offered by the major cloud providers might seem similar; however, some time should be spent gaining familiarity with the available tools to use them efficiently. Other skills like Cloud Computing integration skills, cloud architecture knowledge, cloud management skills are also needed. A certain amount of overhead is required to make the application compatible with the cloud-native infrastructure. The complexity of this code change depends on the constraints set by the cloud provider and the extent to which the offered services diverge from standards; other considerations should also be made to checkpoint the workers' status to some type of cloud storage.

Achieving the proficiency and know-how in the cloud environment will help with **management and organization** and align IT and business strategies. This element is similarly colored in red.

The **End-user experience** also depends on this proficiency; a researcher should also be skilled in cloud tooling and cannot just move from an HPC platform to work with cloud-native tools without undergoing hard training in using the specific platform.

Most major cloud providers offer, for the right price, a mature **LMMA** stack. This is why this element is colored green. As with any cloud-based solution, monitoring the costs of resources consumed by the workloads should be carefully evaluated, and the client should implement prediction models to avoid any exploding expenses.

They also offer cloud security configuration tools; nevertheless, it is still the responsibility of the IT team to configure the services according to their security requirements. Security is the primary concern when running workloads in the cloud, and as such, it is colored in red in our model. Organizational administrators are usually responsible for application-level security configuration, such as necessary access controls for data authorization.

Using a public cloud extends the trust boundary beyond the organization. New risks are introduced by the use of the cloud, like insider threats and the lack of control over security operations[13]. Protecting sensitive data requires encrypting data in

[13] `https://www.nsa.gov/portals/75/documents/what-we-do/cybersecurity/professional-resources/csi-cloud-security-basics.pdf`

transit and when stored at rest. The Management Team must define a robust policy to recognize the data to be encrypted and the proper processes to do it.

3.4.2 Cloud Bursting

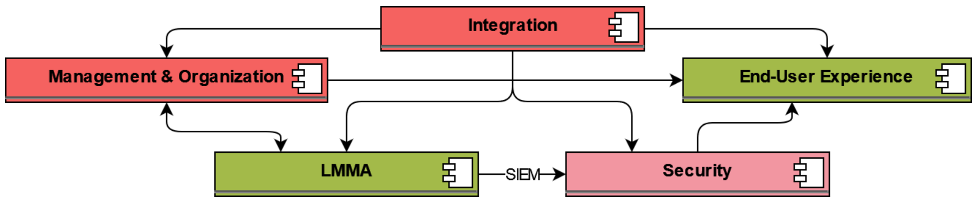


FIGURE 3.15: Administrative Effort of Cloud Bursting

The cloud bursting scenario is modelled in fig. 3.15: a high level of integration skills is still needed to implement the bursting solution effectively. The management and organization overhead is also needed to ensure a secure and cost-effective bursting to the cloud when the resources on-premises are exhausted. The code transformation is also a major pitfall in this scenario. It requires identical virtual machines templates to be used in both environments, a complete mapping of the applications and their dependencies, and a comprehensive view into the providers' existing integration options. This might be doable in the case of enterprise computing; however, in the case of HPC, it is not practically feasible. The end-user experience should be theoretically average since the cloud bursting is usually transparent to the end-users, who will usually launch their job using the standard HPC tools. The LMMA stack is crucial to recognize capacity shortage and burst to the cloud accordingly. Since the solution involves the usage of a public cloud platform, the same considerations apply here; hence, it is colored similarly to what we have seen in the previous section.

3.4.3 HPC Cloud using HPC Technology

In this case, and since we are using the public cloud infrastructure, the same considerations and eventually the same colors apply for the Management, LMMA, and Security aspects. HPC and cloud applications should run with minimal customization, and ease of use is guaranteed because the researcher can use the same exact toolset used in traditional HPC. As such, both elements are colored green as seen in fig. 3.16.

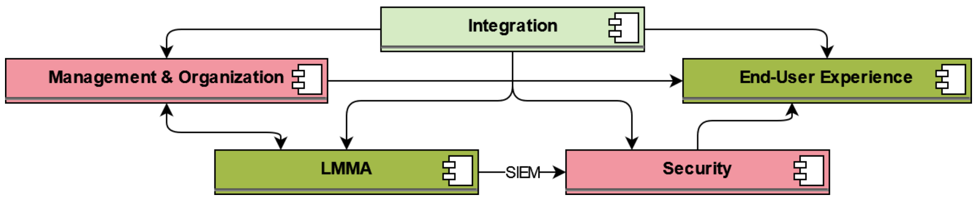


FIGURE 3.16: Administrative Effort of the HPC Cloud using HPC Technology

3.4.4 HPC Grid

Figure 3.17 represents the administrative-effort model of a Grid environment.

The dynamic nature of Grid environments causes challenging administrative overhead. The integration skills are needed to set up the environment. Management issues include Jobs scheduling and the choice of the right LMMA stack. Trust management is also difficult to accommodate since nodes and users are constantly joining and leaving the system. To achieve an acceptable End-user experience, the system should be user-friendly by hiding technological complexity from the end-user. Security is a significant overhead in this case; many aspects like information security, data protection, authorization, and service level security should be considered. Grid systems require resource-specific and system-specific permissions. In a distributed computing project like Folding@home [14], the internal threats are to be carefully evaluated and mitigated.

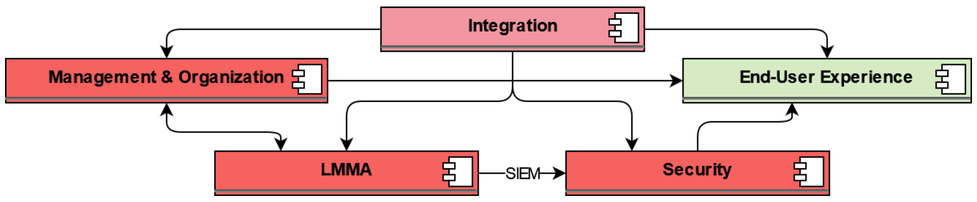


FIGURE 3.17: Administrative Effort of the HPC Grid

3.4.5 Containers in HPC

Using containers in HPC requires a deep understanding of container packaging and orchestration. The choice of the right containerization and orchestration solution plays a critical role, and the management and monitoring of the system will be affected by this choice. Security also plays an important role; supply chain risks and internal threats should be thoroughly considered.

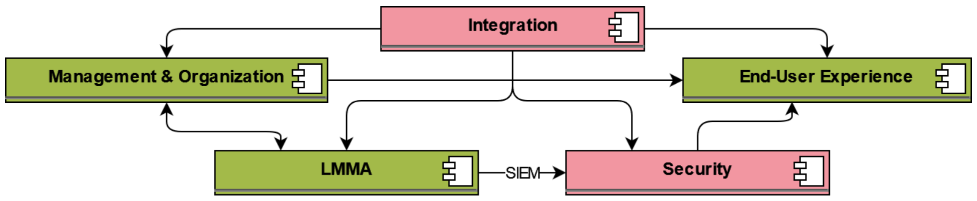


FIGURE 3.18: Administrative Effort concerning the use of Containers in HPC

3.4.6 Cloud Storage in HPC

In this case, the only noticeable overhead is the one introduced by the integration, a certain knowledge of cloud storage is needed; however, we suppose the software vendor has already implemented the interface to communicate with the cloud storage as outlined in section 3.3.6. Furthermore, the code transformation overhead should fade with time, mainly due to the effort towards standardization and the emergence of high-level libraries like the Highly Scalable Data Service (HSDS) (HDF-Group, n.d.), that can use cloud-based storage (e.g., AWS S3) to provide a REST-based service for reading and writing HDF5 data.

[14] `https://foldingathome.org/`

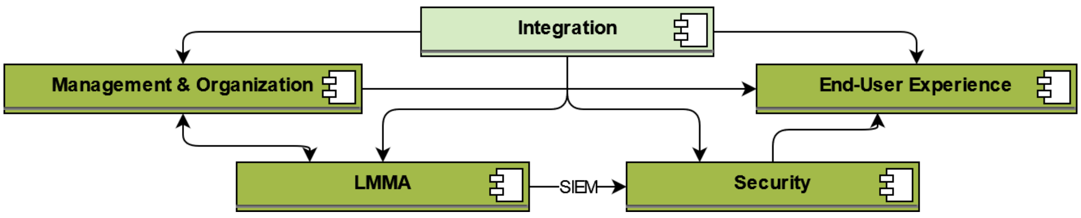


FIGURE 3.19: Administrative Effort concerning the use of Cloud Storage inside HPC

3.5 Cost Efficiency

As seen in section 2.7, a converged system can be on-premises, in the cloud, or a mix of both.

The cost of running workloads on-premises and/or in the cloud must be carefully assessed.

This section presents an economic decision model to quantify the cost efficiency of the converged system; it also helps assess cost changes when bursting or moving entire workflows into the cloud.

To fit both HPC and Cloud, the metric used to estimate the cost is the Total Cost of Ownership (TCO) on a yearly basis. As such, the financial cost model can be easily defined using eq. (3.3)

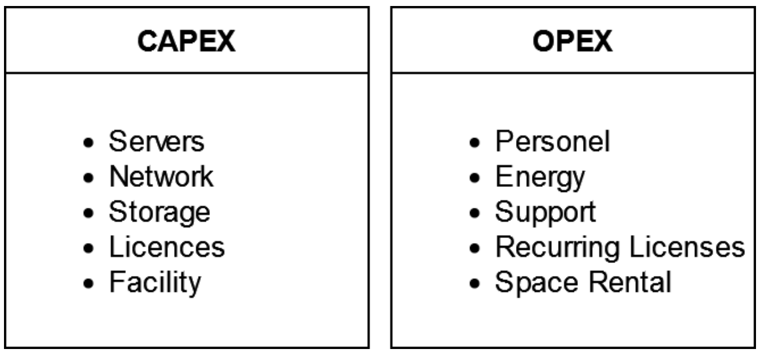
$$Cost_{converged-system} = Cost_{on-premises} + Cost_{cloud} + CoD \tag{3.3}$$

Where $Cost_{on-premises}$ is the cost of the on-premises part of the system, if applicable; $Cost_{cloud}$ is the cloud part cost, if applicable, and CoD is the cost of delay.

Each of these factors is comprehensively explained in the following sections.

3.5.1 Cost On-premises

Figure 3.20 represents the major cost factors in HPC.



CAPEX	OPEX
• Servers • Network • Storage • Licences • Facility	• Personel • Energy • Support • Recurring Licenses • Space Rental

FIGURE 3.20: Main cost factors

A capital expenditure (CapEx) is the money used to purchase, upgrade, or extend the life of an asset. Capital expenditures are long-term investments, meaning the assets purchased have a useful life of more than one year. The CapEx expenses include the following:

- Physical servers costs

- Licenses like the one-time fee for the Operating Systems licenses, Application Software, and management Software licenses.
- Network Cost includes switches, cables, Network Interface Cards (NICs)
- Storage cost includes the expenses for the initial setup of the storage system.

OpEx represents the operating costs, including subscriptions or services needed to put the data center into business use.

- Energy Cost refers to the total power consumption cost, including cooling
- The Facility cost relates to the different equipment needed for operating the data center, like Racks and PDU and the cooling infrastructure.
- Personel Costs represent the cost of salaries paid to employees of the data center and other costs that fall into this category. The recurring license costs include any software license that needs to be renewed, usually on a monthly or yearly basis.

It is quite possible that the investment costs could already include the expenses for the support throughout the system's lifetime – a common practice for procurements done by public data centers. Other cost factors are usually combined into one of these factors; for example, cooling costs are integrated into the energy costs and the cooling infrastructure in the facility costs. Many of the listed costs can be simply divided across the various components of the system. For example, the investment costs for a system are the sum of the costs for the different parts. However, in some cases, the distribution of the expenses across subcomponents is not trivial: For example, when hosting multiple systems into one building, the facility expenses are to be distributed based on the percentage of occupied floor space in order to fairly divide the operational costs like facility and staff costs across IT equipment. Furthermore, the system might experience idle periods. Thus, as these times also incur costs but cannot be assigned to a workload, they must be set to actual workloads. The utilization of a component can be defined as the fraction in its lifetime it is doing valuable work. Since the final utilization is not known apriori, we can use an empiric estimate to adjust for unused resources; the costs for a job can be multiplied with the inverse of the estimated utilization of the component to account for these costs to the job. Supercomputers are usually scheduling production jobs in more than 90% of their lifetime, as seen in fig. 3.21; as such, the usage factor of the compute nodes is, for example, close to one.

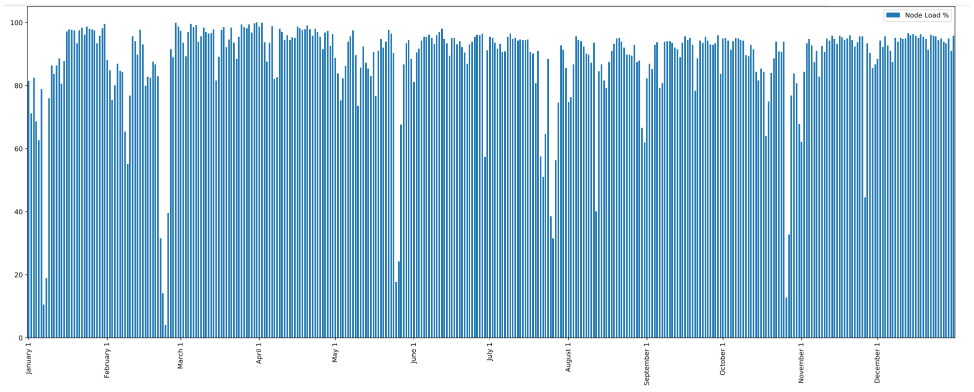


FIGURE 3.21: Mistral nodes load for the year 2020 (Coym, 2021)

Supposing an HPC is usually used to 90 percent and has an economic life of n years, where n is generally around five years. In this period, the system returns more value to the operators than the operation and maintenance costs; after that, the system is deprecated and superseded by new technology in the market.

$$Cost_{on-premises} = \frac{Costs_{CapEx}}{n} + Costs_{OpEx-yearly} \tag{3.4}$$

3.5.2 Cost Cloud

Nowadays, the major cloud providers do have some kind of pricing calculator to estimate the cost of running workloads in the cloud. This covers all of the offerings provided by the cloud provider and related consumed resources like traffic and I/O.

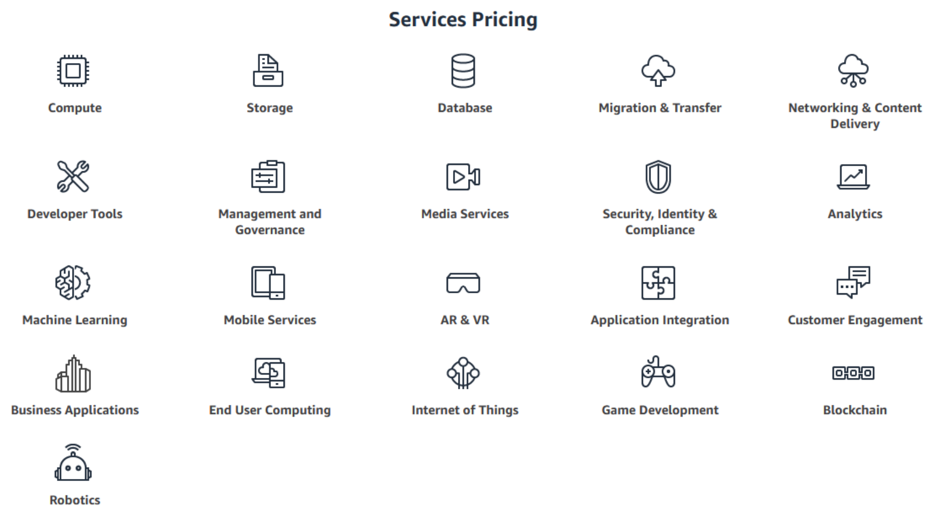


FIGURE 3.22: AWS Paid Service Categories, as found on https://aws.amazon.com/pricing

Figure 3.22 displays the different service categories offered by AWS. Each service is charged differently; for example, for AWS Codestar, the client pays for the used AWS resources (e.g., Amazon EC2 instances, AWS Lambda executions, Amazon Elastic Block Store volumes, or Amazon S3 buckets), in the case of AWS Web Application Firewall, the client is charged based on the number of created web access control lists. Table 3.1 — seen later in section 3.5.5 — outlines a subset of the pricing for the Elastic Compute instances, EC2, offered by AWS.

All cloud providers promise transparency and even offer to lower the bill by using On-Demand, Reserved, or a mix of both pricing models. The provided pricing tools, like https://calculator.aws or https://cloud.google.com/products/calculator/ are reliable and can provide a reasonable estimation of the workload's cost when running in the cloud. However, the user should know precisely the resources used by the application, not only the compute requirements but also the network traffic and I/O requirements which can be extremely expensive.

Hidden costs include the time spent finding the right cloud provider, choosing high-end hardware to use, the design and implementation of the cloud infrastructure without mentioning traffic costs, I/O costs, or just unseen costs, for example, when forgetting to delete unused virtual machine disks.

Some common pitfalls when considering a pure cloud solution are:

- **Traffic Costs**: As already discussed in chapter 1, we should not neglect the cost of the cloud egress traffic. Let's assume we can overcome the data synchronization challenge and send data and receive results seamlessly; however, if the applications produce lots of data that we should pull back to the data center on-premises, the bill will likely increase drastically.
- **I/O Costs**: Cloud providers tend to implement limitations on the I/O for each GB; customers are encouraged to overprovision to gain more I/O for their I/O-intensive application or just buy a certain baseline of dedicated I/O. For example, the cloud cost model presented in (McGough et al., 2014) does not take into consideration the I/O requirement of the application, although dedicated I/O is expensive and is one of the significant bottlenecks when running I/O intensive operations in the cloud.
- **Administrative Costs**: A common misconception is that staffing costs needed to maintain and operate the on-premises infrastructure will fall off when moving to the cloud. This is simply not true; due to the skills needed to manage and maintain the cloud infrastructure, which were covered in section 3.4, and which are at the time of writing extremely on-demand, the staffing costs will likely increase when moving to the cloud.

On the other hand, other factors can lower the cloud usage bill, like the use of spot or preemptible instances, which are instances that use spare compute capacity for less than the On-Demand price, but without any guarantee of continuity, i.e., without interruption. Furthermore, the use of a cloud broker[15] can induce some savings on the cloud bill. A cloud broker acts as a mediator between the organization desiring to purchase the cloud computing service and the cloud vendor. Using his expertise and market knowledge, a cloud broker points to the suitable providers that cover the organization's exact needs and can also help during the negotiation process.

Most cloud providers have a monthly billing period for the used services, commonly referred to as a pay-as-you-go model: eq. (3.5) represents a generalized cost model to calculate the monthly cost of the consumed resources in the cloud, where:

- r represents the consumed resource that can have different types i generally referred to as SKU(Stock Keeping Unit) levels[16].
- R represents the number of all consumed resources during the month m.
- U_i is the unit price for this type i.
- I is the number of the various consumed types or levels for a certain resource r
- n_{ri} is the number of units consumed from resource r from type i. For example, in the case of compute nodes, n_{ri} represents the number of hours where the instance of type i was in use; in the case of traffic or storage usage, n_{ri} represents the GB/GiB used during the billing period.

$$Cost_{cloud-reactive} = \sum_{r=1}^{R}(\sum_{i=1}^{I} n_{ri} \cdot U_i) \tag{3.5}$$

[15] `https://csrc.nist.gov/glossary/term/cloud_broker`
[16] `https://cloud.google.com/skus/?currency=USD&filter=000B-47CC-2924`

As noted, eq. (3.5) can be used to calculate cloud usage costs in the case of a reactive resource allocation based on a pay-as-you-go usage. When considering a proactive resource allocation where a certain usage commitment for a particular resource r is paid in advance- usually on a yearly basis- the cost of the consumed capacity exceeding the usage commitment is considered in the equation alongside the commitment price. The yearly cost of the committed use can be represented using eq. (3.6)

$$Cost_{cloud-proactive} = \sum_{r=1}^{R} Commitments_{yearly} \tag{3.6}$$

Although (Koch, Assunçao, and Netto, 2012) found that a proactive approach is the best way for meeting QoS for a minimum cost, the usage of all the resources **cannot be known in advance**; as such, the resulting **yearly** $Cost_{cloud}$ can be expressed using eq. (3.7)

$$Cost_{cloud-yearly} = \sum_{m=1}^{12} Cost_{cloud-reactive}(m) + Cost_{cloud-proactive} \tag{3.7}$$

Where m is the month number, $Cost_{cloud-reactive}$ and $Cost_{cloud-proactive}$ as defined above.

3.5.3 Cost of Delay

Another factor to consider is the "Cost of Delay"[17]. It refers to the financial impact that a project's delay will have on the institution if not started at a specific date.

The increased interest in the investment for the duration of the delay can significantly affect net profits. The cost increases with the course of the investment; the longer the investment is delayed, the greater the cost is.

One of the easiest ways to calculate the CoD is to multiply the expected daily profit of the project by the number of delayed days. This calculation can get more complex if we include factors like losing clients or reputation, missing the chance to deliver state-of-the-art and prominent research results, and eventually higher labor costs.

3.5.4 Cost Analysis for Different Convergence Solutions

We will now see how our model applies to the different convergence scenarios described in the chapter 2

HPC Cloud using 100% Cloud-native Solutions

In this case, only $Cost_{cloud}$ and CoD are considered. One might argue that there is no delay to start running workloads in the cloud; however, we should consider the software transformation that should take place to adapt the scientific workflows to the cloud tooling. This transformation includes the change of the scheduler and the adoption of cloud storage and concepts. As such, CoD should be considered and quantified to obtain a realistic assessment of the final costs.

As noted previously, most research papers (Smith et al., 2019; McGough et al., 2014; Emeras et al., 2016) on this subject concluded that running scientific workloads on-premises is more economical than running them in the cloud.

[17] `https://en.wikipedia.org/wiki/Cost_of_delay`

We will also address a real-world example in section 3.5.5, particularly by substituting the compute and storage parts of an on-premises cluster with cloud alternatives and inspecting the cost evolution.

Cloud Bursting

One of the main cost concerns, in this case, is the traffic generated by the cloud instances, which need - after a certain runtime - to checkpoint their results to the on-premises storage.

Furthermore, the increased OpEx costs, due to the need for highly qualified personnel in both HPC and Cloud alongside the maintenance and support of two environments instead of just one, highly reduce the cost-effectiveness of this solution.

HPC Cloud using HPC Technology

In this case, $Cost_{converged-system}$ is simply equal to $Cost_{cloud}$ since the CoD can be neglected; the provisioning in the cloud for a reasonably sized cluster should happen without any delays, and the scientific application usually does not require any changes to run in this environment. The same scheduler used in HPC can also be seamlessly used in the HPC Cloud; for example, using the slurm cloud integration, it is possible to launch an auto-scaled Slurm cluster in the cloud. The cluster auto-scales according to job requirements and queue depth, providing the end-users with a set of elastic resources without any waiting time.

HPC Grid

This is one of the most cost-effective solutions since the participating public donates a significant part of computing costs.
The CoD can be significant depending on the type of project, and since most projects using HPC Grids depend on compute resource donation, a delay is expected and can be coped with.
Moreover, the OpEx costs are also kept minimal by using Open Source software and benefiting from the work offered by the enthusiastic contributors.

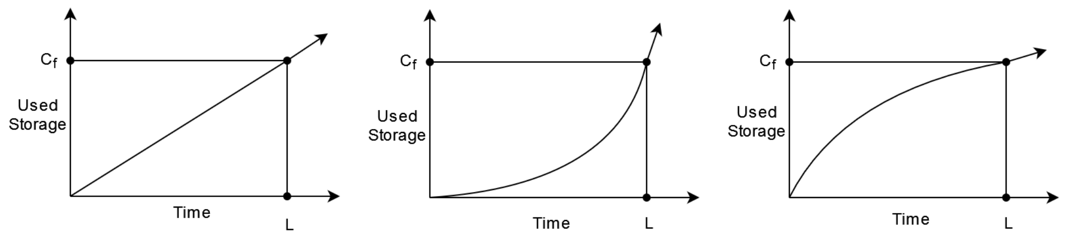
Containers in HPC

CapEx and OpEx costs remain unchanged. CoD should include the time needed to containerize the application. Furthermore, using containers in HPC requires a qualified staff with a deep understanding of container packaging and orchestration. This fact leads to an increase in personnel costs and might introduce new licensing expenses, especially if the containerization solution decides to monetize the use of the software.

Cloud Storage in HPC

Using some type of object storage inside HPC can leverage BigData applications running in HPC. On the other hand, most scientific applications need to be altered to be able to communicate with the object-store. Taking into consideration the feasibility options found in section 3.3.6, we should consider two cases:

Usage of on-demand object storage
Mainly provided by a cloud provider but can also be some provider who places their storage system on-premises and only charge for the used capacity. Supposing an



(A) Linear Evolution (B) Convex Evolution (C) Concave Evolution

FIGURE 3.23: Different Scenarios for the Storage Usage Evolution

on-demand pricing schema, we need to know the monthly evolution of the storage during the system lifetime; we can generally differentiate between three types of evolution:

- An increasing linear distribution, as seen in fig. 3.23a, starting with a newly installed, unused system, until reaching the month "L" representing the maximum lifetime of the system, which corresponds to the final maximal used capacity "C_f".
- An increasing convex distribution as seen in fig. 3.23b, which fosters an on-demand approach to provide the needed storage, since the storage usage for most of the system lifetime is below the baseline and as such a substantial initial investment is not justified.
- An increasing concave distribution as seen in fig. 3.23c, which encourages an on-premises approach and an initial investment in provisioning the whole system, since it will be heavily occupied within the first usage period of the system. Other edge cases, like the presence of some peaks higher than "C_f" for a time "t" lower than L, are practically very close to C_f and would not contradict the recommendation mentioned above.

For the sake of fairness on one hand and simplicity on the other, we will consider a linear distribution to calculate the cost of the on-demand system for the whole lifetime of the system, thus, as depicted by fig. 3.23a. Supposing that the monthly storage unit price U_m for a month m is kept invariable, the final cost of the system is the sum of the used capacity at the end of each month multiplied by this monthly fee.

$$\begin{aligned} Cost_{storage-on-demand} &= \sum_{m=1}^{L} m \cdot \frac{C_f}{L} \cdot U_m \\ &= \frac{L(L+1)}{2} \cdot \frac{C_f}{L} \cdot U_m \\ &= \frac{(L+1)C_f}{2} \cdot U_m \end{aligned} \tag{3.8}$$

As shown in eq. (3.8), the final cost of the on-demand storage will increase linearly with the Lifetime "L", the final used capacity "C_f" or the monthly storage unit price U_m.

The slope of the linear function is represented by:

$$S = \frac{C_f}{L} \tag{3.9}$$

Taking this calculation one step further, we can consider the case of a system initially filled with a certain amount of Data, represented by C_i, which might be the result of a migration of an old decommissioned storage system. Figure 3.23a illustrates this scenario.

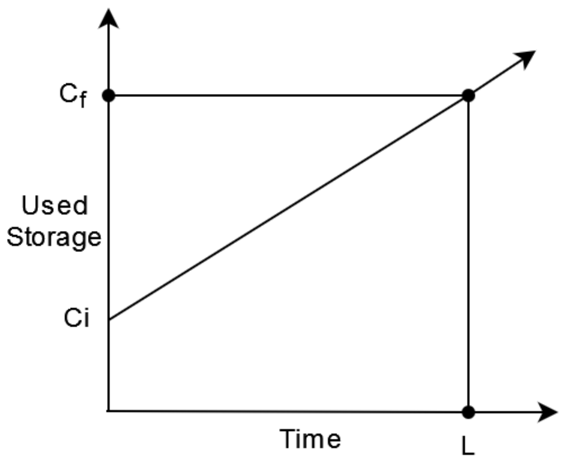


FIGURE 3.24: Linear Storage Evolution with initial Data

Equation (3.10) represents the generalized form from eq. (3.8).

$$\begin{aligned} Cost_{storage-on-demand} &= \sum_{m=1}^{L} m \cdot \frac{(C_f - C_i)}{L} \cdot U_m + \sum_{m=1}^{L} C_i \cdot U_m \\ &= \frac{L(L+1)}{2} \cdot \frac{(C_f - C_i)}{L} \cdot U_m + L \cdot C_i \cdot U_m \\ &= \frac{(L+1)(C_f - C_i)}{2} \cdot U_m + L \cdot C_i \cdot U_m \end{aligned} \tag{3.10}$$

In section 3.5.5, eq. (3.10) will help us to assess a real-world example and provide us with a better insight regarding the use of on-demand object storage versus on-premises provisioned storage.

In this case, the slope of the linear function is represented by:

$$S = \frac{(C_f - C_i)}{L} \tag{3.11}$$

Implementing an object storage solution on-premises The second scenario is the use of object storage on-premises, here we have also two choices:

- **Setup an object storage system on-site**: One can choose between the different providers who specialize in this domain or can implement the system using commodity hardware; most object stores like Ceph and Swift do not require expensive and top performance servers to operate, prooving to be cost-efficient endeavor, the performance of such implementation will be discussed in the coming chapters.

- **Use of already available hardware to provide an Object Storage Interface**: Without a doubt, this is the most cost-effective solution since it simply does not incur any extra costs. An initial personnel investment is to be expected; however, it can be neglected since many open sources solutions like MinIo (MinIO-Inc., n.d.) and Ceph (Weil et al., 2006) offer easy-to-implement tutorials to setup such environments. We will elaborate on the performance of such implementation in the upcoming chapters.

3.5.5 Real World Evaluation: Mistral Case Study

We will now consider a real-world scenario by examining possible convergence scenarios in or around Mistral. To accomplish this evaluation, we use the values related to the cost factors already published in (Lüttgau and Kunkel, 2018) and (Bundesanzeiger, 2020).

Computing Cost Comparison between Mistral and HPC Cloud Solution

A trivial approach for estimating the cost of the computing cost of Mistral[18] in the cloud is to find comparable cloud nodes and perceive how much it would cost to reserve the system for around five to six years, which is the expected lifetime of the computing system and finally compare the results to what the computing part of mistral costs.

At the time of writing, Mistral has around 3300 Nodes, each having two processors from 2 types of processors, Intel Xeon E5-2680 v3 12-core and Intel Xeon CPU E5-2695 v4. Due to active Hyper-Threading, the operating system recognizes two logical CPUs per physical core. As such, around half of the nodes have 48 logical CPUs, and the other half 72.

Due to their alleged competitive pricing schema, we will compare the cost of the above nodes to EC2 instances provided by Amazon. AWS does not offer instances with physical cores but with vCPUs, which are logical CPU cores, with Hyper-Threading enabled, which will ease our comparison. Taking into consideration the network bandwidth and the memory size, we find a comparable instance labeled m5n.12xlarge as seen in table 3.1

Name	Memory	vCPUs	Instance Storage	Network Performance	Linux On Demand cost	Linux Reserved cost
t4g.nano	0.5 GiB	2	EBS only	Up to 5 Gigabit	$0.004200 hourly	$0.002600 hourly
m5n.12xlarge	**192.0 GiB**	**48**	**EBS only**	**50 Gigabit**	**$2.856 hourly**	**$1.799 hourly**
u-12tb1.112xlarge	12288.0 GiB	448	EBS only	100 Gigabit	$109.200000 hourly	$67.305420 hourly

TABLE 3.1: Cost of an AWS M5N EC2 instance as published by `ec2instances.info`

We suppose that we can reserve the system for 6 years, and for a usage of around 90%, as seen in fig. 3.21 ,the computation cost can be calculated as follow:
$Cost_{compute} = 0,9 \cdot 6 \cdot Commitments_{ec2-yearly} = 0,9 \cdot 1,79 \cdot 24 \cdot 365 \cdot 6 \cdot 3300 = 279.424.728$ or $279M.

Let's assume we obtain a significant discount of 50 % for this long-term agreement, we end up with $140M, and this is just for the compute costs without mentioning I/O Costs, storage, and traffic costs.

[18]`https://www.dkrz.de/up/systems/mistral/configuration`

	Investement	**Power Consumption**
Compute	15.75 M	1350 kW
Network	5.25 M	50 kW
Storage	7.5 M	250 kW
Archive	5 M	25 kW
Facility	5 M	

TABLE 3.2: Mistral cost factors

The Mistral compute costs for the same time period are:

$$Cost_{compute} = Cost_{compute\_invest} + Cost_{net\_invest} + Cost_{power} + Cost_{facility} + Cost_{staff} + Cost_{rent}$$
$$= 15.75M + 5.25 + 1400 \cdot 24 \cdot 365 \cdot 6 \cdot 0.25 + 5M + 750K + 250k \cdot 6$$

= 21 M + 18,4 M + 5M + 2.25 M
= €46,65M
= $60M which is much cheaper than the cloud costs.

Note that our calculation is biased to the cloud solution since we neglected the historical price evolution factor; the ec2 pricing at the time of writing is cheaper than in 2015 when Mistral was provisioned; a more detailed study might consider the historical price change of ec2 machines compared to the CPU historical price change, which will definitely increase the cloud costs. Nevertheless, the costs of this heavily used on-premises cluster are much cheaper than a cloud-only solution.

Another factor to consider is the distribution of the jobs on Mistral and the number of nodes used by each job. Table 3.3 shows the job distribution on the different nodes collected during one year. The number of nodes used by each job and the time — in hours — are recorded for each job. Node-hours represent the time multiplied by the number of nodes.

Nodes	**1**	**2**	**3-8**	**9-16**	**17-32**	**33-99**	**100-499**	**500+**
Jobs	3,784,716	2,622,661	378,963	363,198	201,669	84,474	26,297	380
Percent	56.32%	39.03%	5.64%	5.40%	3.00%	1.26%	0.39%	0.01%
Node-hours	716,480	165,220	1,491,662	2,768,859	3,753,182	6,282,016	9,320,025	380,350
Utilization	2.88%	0.66%	6.00%	11.13%	15.09%	25.25%	37.46%	1.53%

TABLE 3.3: Jobs distribution on Mistral as collected by (Coym, 2021) during one year

We notice that more than 50% of the jobs use only one node, and for those types of jobs, it may be economical to use an HPC Cloud solution; however, since the node-hour usage of those single-node jobs is under 3%, this should not affect the final cost of the whole system.

As such, two main factors should be evaluated when considering the use of cloud infrastructure instead of the on-premises one. The first one is the total use of the system, and the second is the distribution of the jobs on the different nodes, mainly how many of them require single or less than five nodes. The use of a cloud solution is advantageous if the usage of the system is low – referring to the calculation done above, this should be less than 20% – or if the percentage of node-hour consumed by jobs, using a single or a limited number of nodes, is higher.

Cost Comparison between Tape Archive and an On-demand Object Storage

The DKRZ[19] uses seven StorageTek SL8500 systems with on-premises library units and an additional unit as an offsite backup in Garching. The tape technology is supposed to remain usable for around two to three generations of supercomputers, and this might be due to two main reasons:

- The current configuration does not consume all the available drive slots.
- As tape media increases in capacity with each new generation of tape media, there is the potential to reclaim a significant number of tape slots.

Each SL8500 module, equipped with about 20 drives, costs slightly under €1M, thus €7M for the whole system, but since we can use it for three generations, we can reduce the investment costs to €2.3M for the lifetime of one supercomputer. Adding the costs of the license for HPSS, the storage cache, and the support for the library and the software, we obtain a total investment of €7.5M for the system's lifetime. Maintenance and support of the system require three specialists, and since the annual operational budget for DKRZ is about €3M , we assume the three staff are 5% of that total leading to a yearly cost of €150k or €750k for five years. Furthermore, the floor space to hold the system is equivalent to €70k per year or €350k in five years. The above yields around €8.6M for the lifetime of the system. The total system offers 67,000 tape slots; each tape medium has 2.5 TByte of raw capacity, costs around 20€, and can be used for five years leading to a total cost of 134 K€. By adding the above values, we find that the cost of the tape archive for the system's lifetime is €8,7M or around $10,5M [20] with a total used capacity of 167,5 PB.

To calculate the cost of an alternative on-demand storage archive system for the lifetime of Mistral,i.e., $Cost_{storage-on-demand}$, we use eq. (3.10), for this purpose, we need to determine the initial amount of data available on the archive system when Mistral was put in service in mid-2015, we refer to this value as C_i=28.5 PB.

The values of C_f cannot be easily predicted in advance and we must rely on the historical data to obtain it.

Having L=60 and C_f around 101 PB, and supposing the archive was filled lineary similar to fig. 3.24, we still need to determine U_m.

Since U_m depends on the cloud storage provider, we try to find some providers who do not charge for traffic cost, for example, wasabi [21] displays a price tag of 5.99 Dollar/TB/Month without any Egress costs, while offering hot on-demand storage where the client has instant access to the data whenever needed regardless of its use case, i.e., backup, disaster recovery, or long-term archiving. We neglect the price of the internet connectivity for two reasons:

- Most HPC data centers nowadays have an excellent internet connection to a highly peered backbone, like most cloud providers.
- The possibility of using an on-site on-demand system as stated above can also be considered.

For L=60, C_i=28.5, C_f=101.6 and U_m = 5.99 \$, eq. (3.10) yields to:
$Cost_{storage-on-demand}$ = \$23.45M

[19] Deutsches Klimarechenzentrum: German Climate Computing Center
[20] `xe.com`
[21] `wasabi.com`

This is around two times the cost of the in-house system. Even with the expected discount of 20% usually obtainable in the cloud world, the cost of the on-demand system is still higher than the on-premises implemented solution.

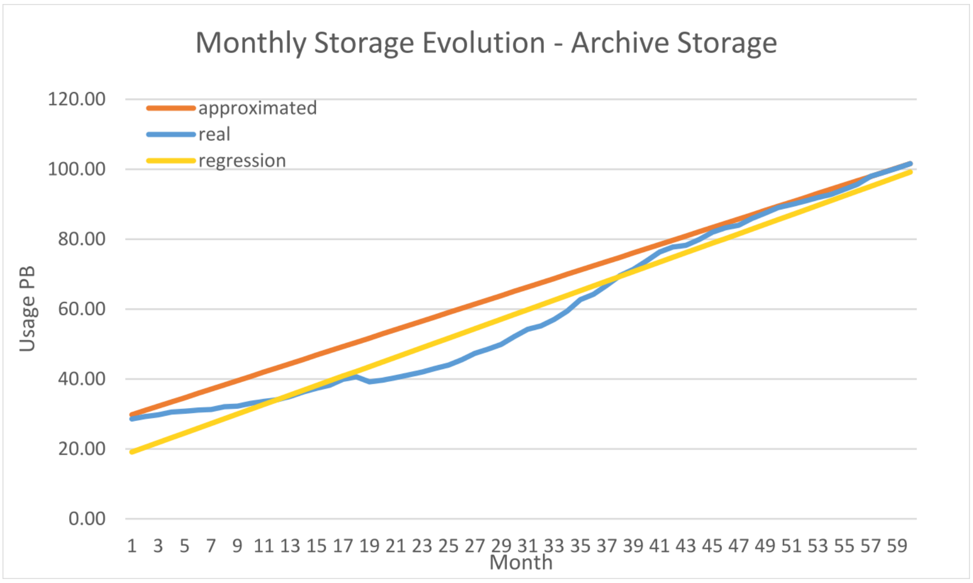


FIGURE 3.25: Archive system monthly storage evolution

Since the usage of the DKRZ archive system is recorded for auditing purposes, the monthly usage was obtained and used to plot Figure 3.25 for the lifetime of Mistral; we notice the following:

- The first month is July 2015, when Mistral was put in service.
- The approximated values are obtained when supposing that the filesystem was occupied linearly similar to fig. 3.24. We already know the values from L, C_f and C_i from the above calculation; we can use them to calculate the slope S using eq. (3.11). From the historical data to obtain them, we have: $C_i = 28.5PB$ and $C_f = 101.6PB$, we get $S \approx 1.22$ and use it to draw the approximated linear evolution.
- Using Excel or some other statistical tool, we conduct a linear regression analysis to discover the relation between the storage usage as output values and the month number as input values, and to demonstrate the validity of the linear approximation described above. The signification F value obtained is equal to 6.77895E-44 and is extremely lower than the threshold value of 0.05, which means that our approximation approach is precise. The values obtained from this regression are plotted, and we can see that, after a short time, they converge with the approximated values.
- The exact value of $Cost_{storage-on-demand}$ is found equal to \$21.11M; we can calculate the error rate of our estimation accordingly, and we find it is around 9%.

Cost Comparison between Parallel Filesystem and On-demand Object Store

The initial investment in the Lustre parallel filesystem was €7.5M, with a total capacity of 54 PB. The power consumption is around 250 kWh and the cost of 1kWh is 0,25 € leading to $250 \cdot 0,25 \cdot 24 \cdot 360 \cdot 5 = 2.7M$ for 5 years. Similar to section 3.5.5 the staff costs around €0.75M in five years. We suppose that the maintenance and support costs are included in the initial investment; the costs of the systems sum up to around €11M or about $13.2M.

The exact PFS[22] usage of the Mistral is also obtained and used to plot fig. 3.26; we notice the following:

- The approximated values are obtained when supposing that the filesystem was occupied linearly; this should compromise for the usage peeks during the systems lifetime and the fact that, due to performance reasons, the used storage capacity should not exceed 90% of the total achievable capacity. We already know that L=60, we sill need C_f to calculate the slope S using eq. (3.9). As mentioned above, since the 90% threshold should never be exceeded, we can determine the value of C_f apriori in relation to the total capacity C_t: $C_f = 0.9 \cdot C_t$ As such, $C_f = 48.6$ and $S \approx 0.82$

- Using Excel or any other statistical tool, we run a linear regression analysis to discover the relation between the storage usage as output values and the month number as input values, and to demonstrate the validity of the linear approximation described above. The signification F value obtained is equal to 5.21153E-28 and is extremely lower than the threshold value of 0.05, which means that our approximation approach is accurate. The values obtained from this regression are plotted, and we can see that they are very close to the approximated values.

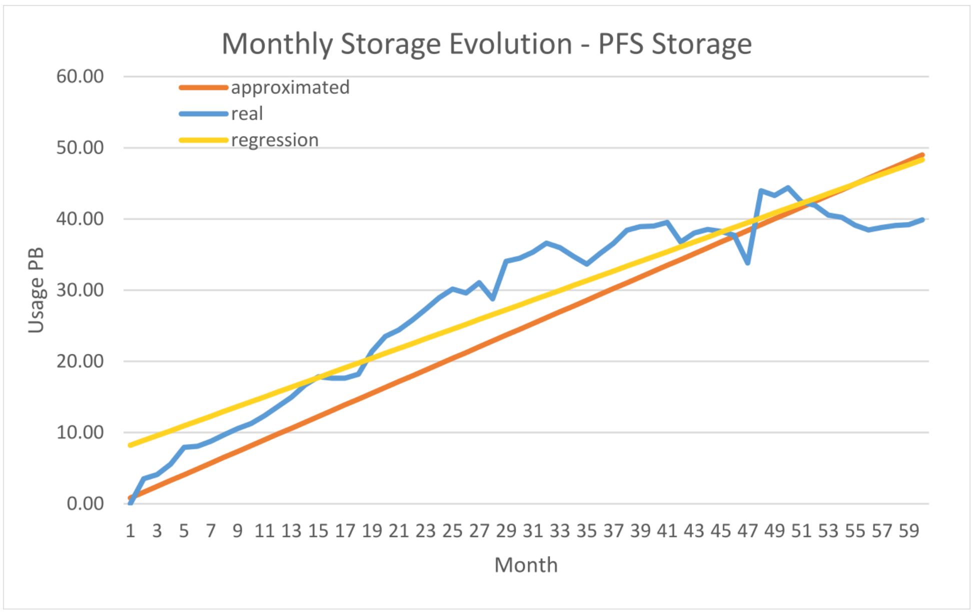


FIGURE 3.26: Parallel filesystem monthly storage evolution

Having L=60, U_m=5.95 and C_f and using eq. (3.8), we find that the estimated $Cost_{storage-on-demand}$ is around $9M, the exact value is $10.09M, leading to an error rate of around 10%. Both values are cheaper than the on-premises implemented solution.

[22]Parallel File System

3.6 Conclusion

All the components of our assessment model from fig. 3.2 have been thoroughly examined.
Figure 3.27 summarizes the full picture of our model.

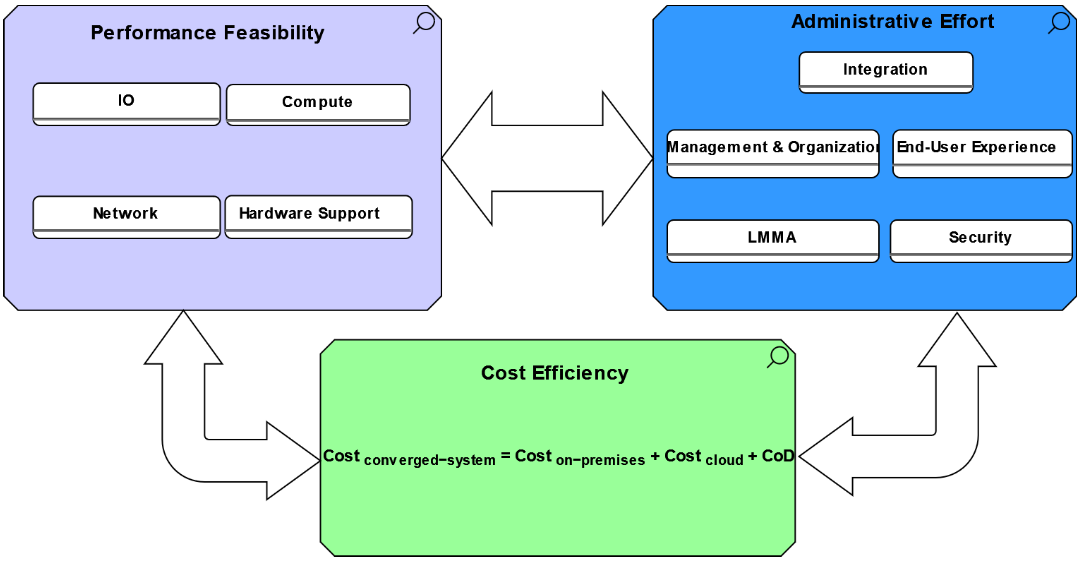


FIGURE 3.27: HPC Cloud Convergence Assessment Model

Table 3.4 illustrates the comparison between the different scenarios using the full-scaled assessment model; the negative signs depict a negative evolution in the sense of high overhead or high costs, the positive signs depict a positive evolution in the sense of a low overhead or low costs.

Scenario	Performance Feasibility				Administrative Effort					Cost Efficiency		
	I/O	*Comp*	*Net*	*Hardw Sup.*	*Integ.*	*Mgmt*	*End-User Exp*	*LMMA*	*Sec*	*Cost_onprem*	*Cost_cloud*	*CoD*
HPC Cloud native	-	++	-	++	-	-	-	++	-	NA	-	+
Cloud Bursting	- -	+	-	-	-	- -	+	++	-	+	-	-
HPC Cloud-HPC	-	++	++	++	++	-	++	++	-	NA	- -	+
HPC Grid	- -	+	- -	- -	- -	- -	+	-	- - -	+++	+	-
Containers in HPC	+	+++	++	+	-	+	++	+	-	++	NA	-
Cloud storage HPC	+	+++	+++	+++	+	++	++	++	++	++	+	+

TABLE 3.4: Comparison of the different scenarios using the assessment model

We notice that some solutions like **Cloud Bursting** are technically feasible but might be less I/O performant, and can introduce a considerable administrative effort while producing high costs, as seen in section 3.5.4. Other solutions like **HPC Grid** have a better feasibility possibility and are highly cost-effective but produce a substantial administrative effort without any guarantee for performance stability.

Convergence approaches like **HPC Cloud using 100% Cloud-native Solutions** are generally feasible but require a deep knowledge of the Cloud environment and introduce new security risks leading to a high administrative effort. The cost is also extremely high for dense HPC jobs requiring a vast number of first-class nodes, as seen in the real-world evaluation in section 3.5.5. A more feasible solution is the **HPC Cloud using HPC Technology** where scientific workloads can be run seamlessly without any delay or any code transformation. Scientific and Big Data workflows are guaranteed to work in this environment while presenting a known toolset for the end-user launching those applications. However, similar to the previous approach, the cost for heavy HPC jobs requiring a large number of well-fitted nodes is surely to be a costly endeavor, even if huge discounts are provided as outlined in section 3.5.5.

The performance feasibility of **Containers in HPC** is mainly positive, and although a certain administrative effort is expected, the setup cost and maintenance of this solution are also acceptable.

One of the most promising solutions is the use of **Cloud Storage in HPC**. The administrative effort is kept moderate once the application's compatibility with the cloud storage is provided. From the cost aspect, it can be even cheaper than available HPC filesystems as seen in section 3.5.5. The performance feasibility is still to be thoroughly investigated; this is why we focus in the upcoming chapters on the performance of object storage inside HPC:

Since cloud storage uses the HTTP protocol for the communication between the client and the storage server, we start by carefully investigating its performance inside HPC in chapter 4. In chapter 4, we also compare the performance of the different and relatively new versions of HTTP protocols.

Chapter 5 presents our approach to quantify the performance of the S3 API by extending commonly used HPC I/O benchmarks and launching them against on-premises S3 implementations inside an HPC Cluster, Mistral, and against different S3 Cloud providers as well. We also try to overcome the performance and scalability bottlenecks by systematically replacing parts of a popular S3 client library with lightweight replacements of lower stack components. Launching the extended benchmarks against the introduced S3Embedded library should prove its high scalability and performance and demonstrate the possibility of integrating applications with both cloud and traditional POSIX filesystems.

3.7 Summary

This chapter presented the research methodology used in the rest of this work. An assessment model was also introduced to assess the degree of HPC and Cloud convergence. Using this model, we compared the different solutions presented in chapter 2 and came to the conclusion that using object storage inside HPC is one of the most promising approaches to thrive the HPC Cloud convergence. We also examined the cost of running workloads and storing their results on-premises and in the cloud in section 3.5.5. In the next chapter, we will address the performance overhead introduced by using cloud storage inside HPC by thoroughly analyzing the overhead of REST inside HPC.

Chapter 4

Overhead of REST on HPC Systems

In chapter 3, we found that the use of cloud storage in HPC scored well in the assessment model. This solution can also be cost-effective, as we saw in section 3.5.5. Since cloud storage uses REST/HTTP as a communication protocol (Fielding, 2008), this chapter examines its performance by investigating the REST/HTTP protocol's overhead compared to the HPC-native communication protocol MPI when storing and retrieving objects. The structure of this chapter is a follows: After a short introduction in section 4.1, section 4.2 presents the approach used for modeling the impact of data transfer using measurable performance metrics; it also describes the test scenarios and defines the relevant metrics addressed using our benchmarks. Section 4.3 outlines the experimental procedure, the used systems, and the methodology of the evaluation conducted in this work. Section 4.4 presents the analysis of the obtained results and the model validation by comparing the results obtained for REST and MPI, on two different cluster systems, with the model predictions. A comparison of the performance of the different HTTP protocols on HPC is also provided in section 4.4.1. The last section 4.5 summarizes our findings.

Most of the content of this chapter has been published under (Gadban, Kunkel, and Ludwig, 2020)

4.1 Forword

An overview of object-storage was provided in section 2.4.2: It organizes information into containers of flexible sizes, referred to as objects. Each object holds the data itself as well as its related metadata and has a unique identifier used to locate it, rather than a file name and path. Thus, object storage is highly scalable by design and is basically different from traditional block or file storage systems.

Cloud storage implements an object storage architecture and enables the client applications to directly access the objects using a RESTful API (Richardson and Ruby, 2008). Therefore, any comprehensive performance study of cloud storage within HPC should consider the overhead introduced by the REST system, where the communication takes place over the HTTP protocol (Fielding, 2008).

Many studies (Chen et al., 2017; Zhang et al., 2016) point out the factors behind the latency like the actual hardware used, the request parsing mechanism, and the way communication is performed, e.g., using system calls and supportive libraries Epoll, libevent, or user-space communication.

Many researchers tried to solve data transfer issues through HTTP: Some (Ko et al., 2012) proposed encapsulating TCP data in UDP payloads, others (Devresse and Furano, 2014) proposed a dynamic connection pool implemented by using the HTTP Keep-Alive feature to maximize the usage of open TCP connections and minimize the effect of the TCP slow start. Intel®is marketing DAOS (Lofstead et al., 2016) as the ultimate Open Source Object Store, nonetheless with a high vendor Lock-in potential since the promised performance can only be achieved on its own proprietary Optane (Wu, Arpaci-Dusseau, and Arpaci-Dusseau, 2019) storage Hardware. On the other hand, (Borzemski and Starczewski, 2009) researched the performance of HTTP between geographically dispersed nodes.

Few researchers tried to assess the performance of a REST service on HPC, i.e., within a high-performance network: Since Infiniband (Association, 2020) is one of the most commonly used interconnects in HPC, the performance of IP over Infiniband (Grant, Balaji, and Afsahi, 2010; Bortolotti et al., 2011; Yang et al., 2019; Zhang et al., 2021) has been thoroughly studied; however, the performance of HTTP over IP over IB did not get much attention. Hence, the following section explains our approach to model and assess the performance of HTTP on HPC and analyze the viability of HTTP over Infiniband using hardware performance counters[1] and network metrics.

4.2 Methodology

Assuming that typical HPC applications require maximum throughput and minimum latency, we address in this chapter many questions revolving around the suitability of a RESTful service in HPC and the main factors affecting its performance and its resource consumption.

The two major efficiency indicators addressed in our study are **latency** and **throughput**.

The hardware components, their interconnection, and the software stack available to HPC applications are usually trimmed for performance compared to those serving typical web applications.

Therefore we introduce in section 4.2.1 a modeling approach, based on performance counters, to evaluate the overhead introduced by HTTP when used to provide the communication between two entities consisting of a content server and a client application consuming the content.

The tools and the accomplished tests are described in section 4.3.2. In section 4.3.3 and section 4.3.4, we analyze the variation of latency and throughput, respectively, when changing factors like the number of requests or the file size.

Our benchmark for storage access emulates a best-case scenario (HTTP GET Operation / Read Only Scenario from a "remote" Storage Server) because we only want to test the viability and base performance of REST/HTTP as an enabling technology for an object-store. The introduced model can nevertheless be extended to assess and measure the resource consumption of different object storage implementations. In this chapter, we only focus on the performance of the HTTP protocol inside an HPC environment; the performance of different object storage implementations are addressed in chapter 5.

Comparing the REST protocol to the Message Passing Interface MPI (The MPI Forum, 1993), which is an established data transfer approach in the HPC world, is accomplished in section 4.3.6. To identify the major factors impacting the performance, we vary the underlying hardware and the connection mechanism (Ethernet, Infiniband, RDMA) between the server and the client.

[1]Set of special-purpose registers built into modern microprocessors to gather statistics about performance properties of code execution and data transfer.

The results obtained from these experiments are then used to validate our defined model by comparing them to the predicted ones.

We also compare the different HTTP protocols on HPC in section 4.4.1, before wrapping up this chapter.

4.2.1 Performance Model

To define our performance model, we consider different metrics, which depict the used hardware (CPU, Bandwidth ...), the software stack, and the network protocol in use. Alongside the standard network metrics, we focus on hardware counters of the CPU, namely the number of required CPU cycles to identify the processing cost of a data transfer and the L3 evicted memory to quantify the memory transfer overhead and check the memory efficiency of the different implementations.

In a first step, we consider TCP as a transport protocol; nevertheless, the model is later extended in section 4.4 to cover MPI over different implementations.

The metrics involved can be summarized as follows.

Fixed system parameters:

- R: CPU clock rate in Hz; Rs and Rc correspond to the server and client, respectively.
- rtt: round trip time or RTT[2].
- mtu: maximum transfer unit.
- mss: maximum segment size or MSS[3], transmission protocol-dependent (see eq. (4.4))
- mem_tp: the memory throughput, i.e., the speed of data eviction from L3 to the main memory; influenced by the speed of the different caches
- eBW (Chang and Thomas, 1995): is the effective bandwidth between client and server, which is the minimum bandwidth of all the hardware components between them (memory throughput, PCI Bus, network interfaces...)

Experiment-specific configurations:

- Obj_size: file size transferred from the server to be read by the client.
- Nreq: number of requests achieved in 60 sec.
- Ncon: number of connections kept open.
- Nthr: number of CPU threads executing the benchmark on the client.

Observable metrics (e.g., using Likwid):

- CUC: to express the performance metric CPU_CLK_UNHALTED_CORE, which represents the number of core cycles when the core is not halted, CUCs, and CUCc correspond to the server and client, respectively.
- L3EV: the amount of data volume loaded and evicted from/to L3 from the perspective of CPU cores, i.e., the data flowing through L3 (Gruber, 2020; Intel, 2020), L3EVs and L3EVc correspond to the server and client respectively.

[2]Round Trip Time

[3]Maximum Segment Size

- PLR: the packet loss rate, which is theoretically proportional to the number of parallel connections. In fact, the more parallel connections we have, the higher the packet loss rate will be.

In our preliminary model t(request) is the time elapsed between sending the first byte of the request and when the complete response is received; it can be calculated as follows:

$$t(request) = t(client) + t(network) + t(server) \tag{4.1}$$

where t(client), t(network), t(server) are the time fractions needed by the client, network and server respectively to accomplish the request:

$$t(client) = t(compute) + t(memory) + t(cpu\_client\_busy) \tag{4.2}$$

$$t(server) = t(compute) + t(memory) + t(cpu\_server\_busy) + t(pending) \tag{4.3}$$

As a rough estimation of the maximal network throughput when using the TCP protocol, referred to as net_tp, and based on the Mathis et al. formula (He, Dovrolis, and Ammar, 2007), while presuming that the TCP window is optimally configured, we can safely assume that:

$$net\_tp = \min\{\frac{mss \cdot C}{RTT \cdot \sqrt{PLR}}, eBW\} \tag{4.4}$$

where C=1 and mss = mtu-40 in case of Ethernet. From this we can calculate t(network):

$$t(network) = Obj\_size/net\_tp + t\_queuing \tag{4.5}$$

Where t_queuing is the time that the packets spend waiting in a specific buffer for processing at the router or the network interface level. For the sake of simplicity, we suppose that the routing devices between the nodes do not add any latency, and as such, we can neglect t_queuing.

The execution time t(compute) can be defined as:

$$t(compute) = CUC/R \tag{4.6}$$

t(memory) is the time to traverse the different memory caches, usually narrowed down to:

$$t(memory) = L3EV/mem\_tp \tag{4.7}$$

Putting it all together, and in the case of intra-node communication, we can safely assume that:

$$t(request) = \frac{CUCs}{Rs} + \frac{L3EVs}{mem\_tp} + \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + \frac{Obj\_size}{net\_tp} \tag{4.8}$$

Generalizing a bit further, we end up with :

$$t(request) = \alpha \cdot rtt + \beta_1 \cdot \frac{CUCs}{Rs} + \beta_2 \cdot \frac{L3EVs}{mem\_tp} + \beta_3 \cdot \frac{CUCc}{Rc} + \beta_4 \cdot \frac{L3EVc}{mem\_tp} + \beta_5 \cdot \frac{Obj\_size}{net\_tp} \tag{4.9}$$

Where α is a weighting factor ($0 \leq \alpha < 1$) (IETF, 2011), β_i are platform and protocol dependent factors to be evaluated in a later section.

Hence, many factors can influence the above, starting from the application delivering the content, which affects server CPU and memory usage; those metrics are also affected by the type of client consuming the data and the networking protocol in use, and the path traversed by the data.

In the following sections, we validate this model while comparing the performance of HTTP over different types of hardware and connection protocols.

4.3 Evaluation

The tests are performed on two different hardware platforms: the first one is the WR Cluster, a small test system available at the University of Hamburg, the second one is the Mistral supercomputer (DKRZ, 2020), the HPC system for earth-system research found at the German Climate Computing Center (DKRZ).

4.3.1 Test Environments

The nodes of the WR Cluster are equipped with two Intel Xeon 5650 processors, each offering six cores. Those processors operate at a nominal frequency of 2.66GHz and utilize three memory channels. Hyper-threading is enabled on this cluster, and, thus, 24 logical processors are visible in Linux.

The supercomputer Mistral (DKRZ, 2020) provides 3000 compute nodes, each equipped with an FDR Infiniband interconnect and a Lustre storage system with 54 PByte capacity distributed across two File systems. The nodes used for the testing are equipped with two Intel Broadwell processors (E5-2680 @2.5 GHz).

4.3.2 Benchmark and Analysis Tools

The RESTful API is the typical way to realize access to cloud storage, and as such, the tools used in this article were preliminary developed to assess HTTP performance.

The first experiment checks the latency introduced by a simple web server serving static files; the setup, shown in fig. 4.1, consists of the Lighttpd web server (Kneschke, 2020) running on one node and hosting files of different sizes. These files are initially placed in the in-memory (tmpfs) file system to minimize storage-related overhead such as disk drive access time. The tests are conducted on another node using the wrk2 tool (Tene, 2020) which is a standard HTTP load testing tool. wrk2 pretends to deliver accurate latency measurements by avoiding Coordinated Omission (Tene, 2020), in the sense that it measures response latency from the time the transmission should have occurred - according to the constant throughput configured for the run- to the moment it was received.

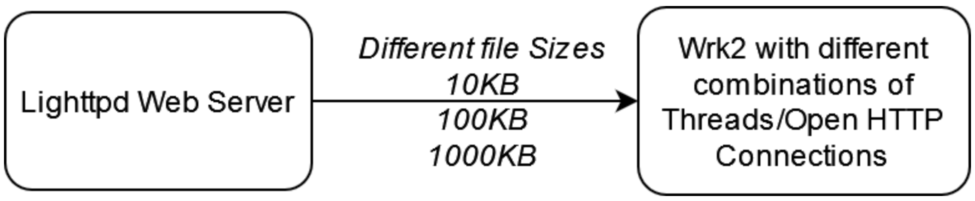

Lighttpd Web Server
Different file Sizes
10KB
100KB
1000KB
Wrk2 with different combinations of Threads/Open HTTP Connections

FIGURE 4.1: A simplified overview of the Benchmark Setup

In the analysis, we vary the number of threads, and the number of HTTP connections kept open while trying to keep a steady rate of 2000 requests/second for 60 seconds for each file size.

4.3.3 Latency

The diagrams in figs. 4.2 and 4.3 show the obtained latency distributions when varying the number of open connections while keeping a fixed file size: As depicted by the chart legend, the first number representing the number of open connections is varying, while the second number denoting the file size is kept stable.

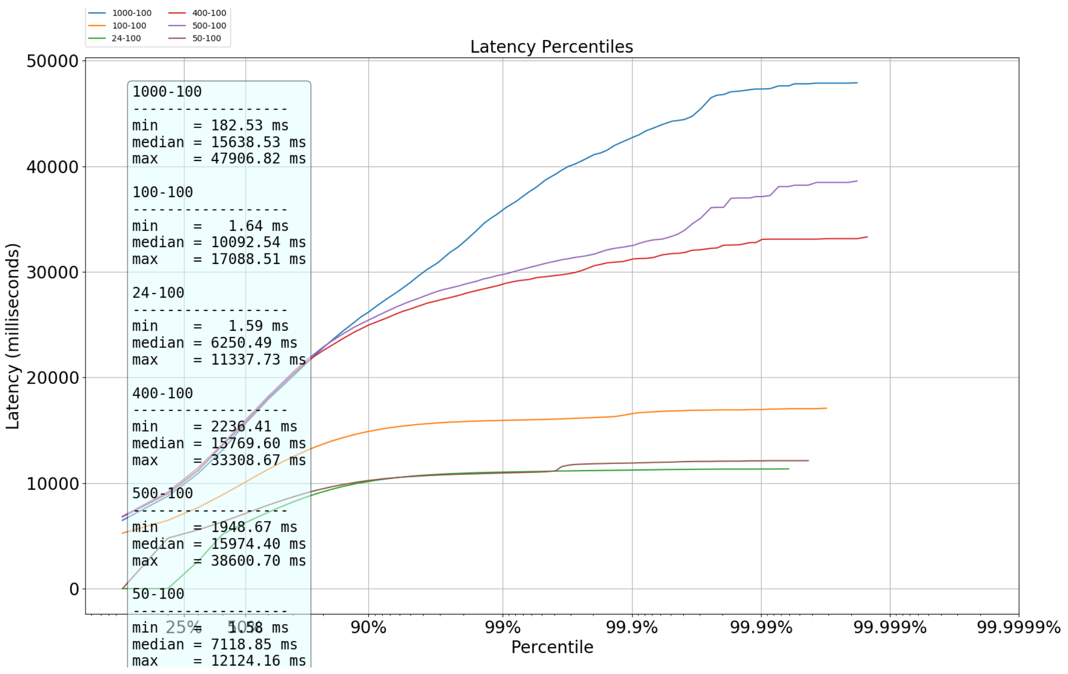


FIGURE 4.2: Latency variation in relation to open connections for a file of size 100 KB

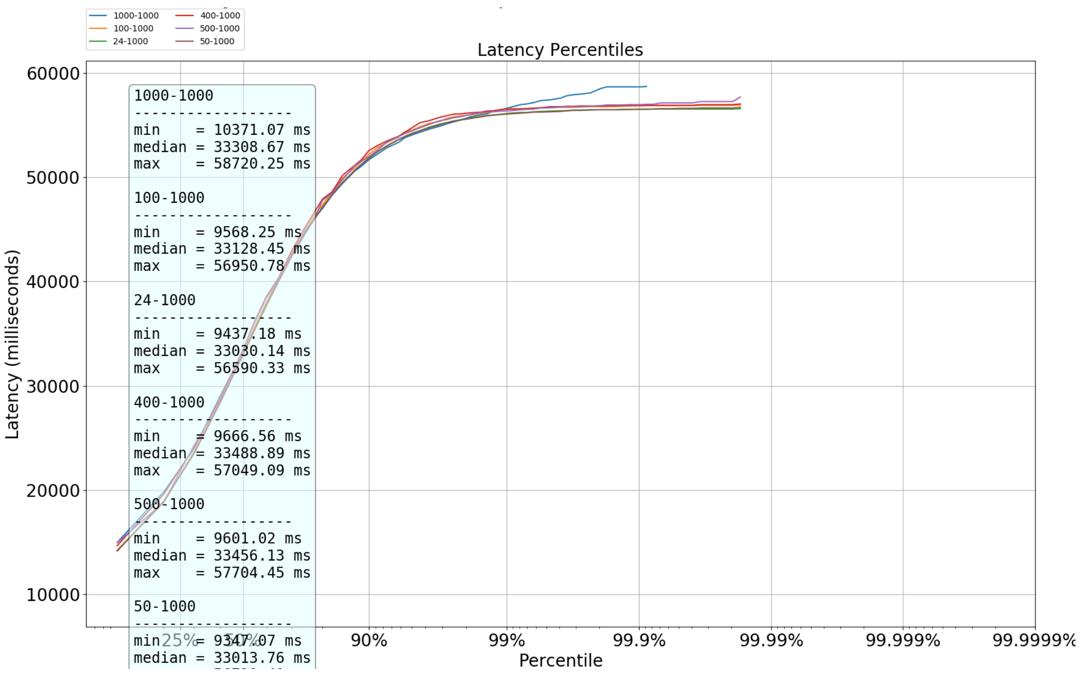


FIGURE 4.3: Latency variation in relation to open connections for a file of size 1000 KB

The diagrams in figs. 4.4 and 4.5 show the obtained latency distributions, when varying the file size while keeping a fixed number of open connections. This time and as represented by the chart legend, the first number illustrating the number of open connections is preserved stable, while the second number denoting the file size varies.

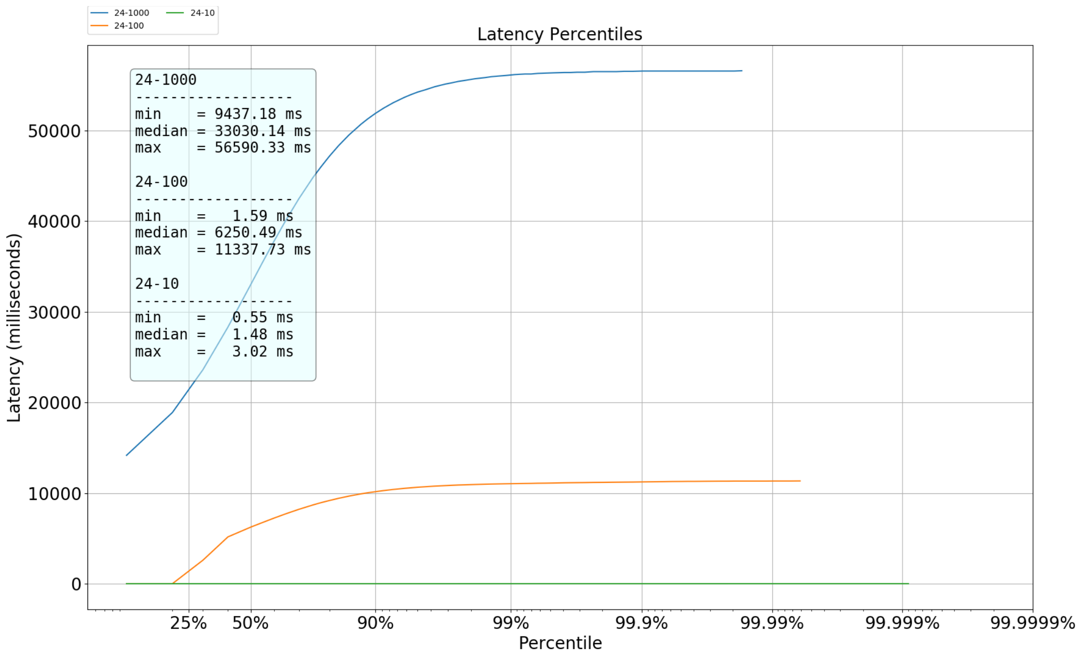


FIGURE 4.4: Latency Variation in relation to file size for 24 open connections

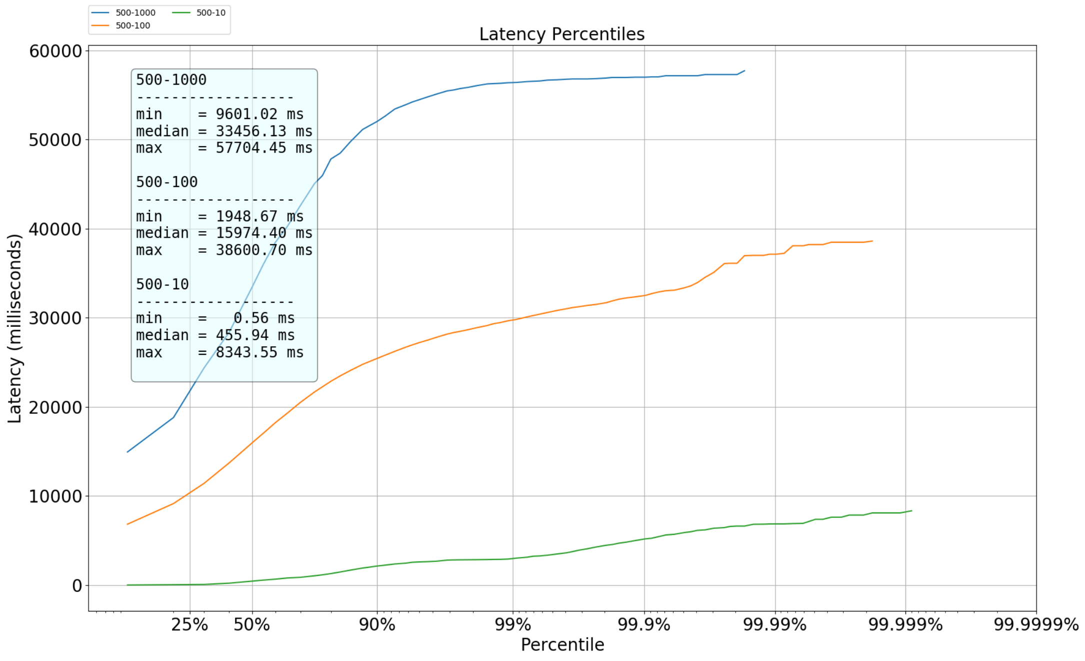


FIGURE 4.5: Latency Variation in relation to file size for 500 open connections

Observations and interpretation:

- Latency does not depend on the duration of the experiment; moreover, it linearly increases with the number of open connections (see fig. 4.2). The higher the number of connections kept open, the higher the chance of packet loss leading to the activation of the TCP flow control mechanism and eventual data retransmission, which causes the increase in latency. This is especially true for small file sizes; however, when the file size grows above a specific limit, the number of connections will become irrelevant to the introduced latency, as seen in fig. 4.3.
- As shown in fig. 4.4, for small file sizes, we observe a latency divergence in particular in the 99 percentile area. For bigger file size, we notice that, in the case of the 100 KB, the desired request rate of 2000 req/s is not met due to the limitation of the underlying network infrastructure, which offers a bandwidth of 1Gb/s, i.e. around 125 MB/s.
- It is interesting to note that, in relation to the file size and as shown in fig. 4.4), larger files lead to higher memory and network latencies in a way that they can saturate the server's network bandwidth, lowering throughput (see fig. 4.3). Therefore, high network bandwidth is more important than compute resources for serving large files. On the other hand, increasing the number of open connections, as seen in fig. 4.5 will trigger TCP's congestion mechanism, leading to a state where the different connections compete for the same bandwidth—increasing the file size as well will cause the open connections to lose packets and get stuck waiting for retransmissions.

A similar latency distribution is also observed when conducting the tests on the same machine, thus using the optimized (Dumazet, 2012) loopback interface, where theoretically the network stack overhead is kept at its minimum, and the mtu is explicitly set to 64K to allow TCP stack to build larger frames. However, although tests using iperf (NLANR/DAST, 2020) yielded a throughput of around 20 Gbps, the maximal throughput achieved using our benchmark is only 500MB/s or 4 Gbps; this is mainly due to the fact that the server and the client were competing on the same resource pool since wrk2 is started with 24 threads (the max for each node).

From these experiments, we learn that to optimize the throughput; the web requests should not be using different open connections but instead use one or a relatively small number of open connections and label the web requests accordingly, a technique commonly known as HTTP multiplexing (Gettys, 1998),already illustrated by fig. 2.9: By using the same TCP connection, multiple HTTP requests are divided into frames, assigned a unique ID called stream ID and then sent asynchronously, the server receives the frames and arranges them according to their stream ID and also responds asynchronously; same arrangement process happens at the client-side allowing to achieve maximum parallelism.

4.3.4 Throughput

The network throughput of our system is proportional to the number of requests. It is calculated as follow:

$$Throughput = \frac{Nreq \cdot Obj\_size}{time}$$

In our tests, the benchmark ran for a time of 60 seconds. Hence, comparing the achievable throughput for the different connections/threads combinations is the same as comparing the achieved number of requests.

Figure 4.6 shows that, in the case of inter-node communication, an increase in the number of open connections will increase the throughput; on the other side, an increase in the number of threads yields the same effect. However, for file sizes above 1 MB, the influence becomes negligible, and the increase in the number of threads/open connections raises the congestion/loss rate, causing the benchmark to return different errors.

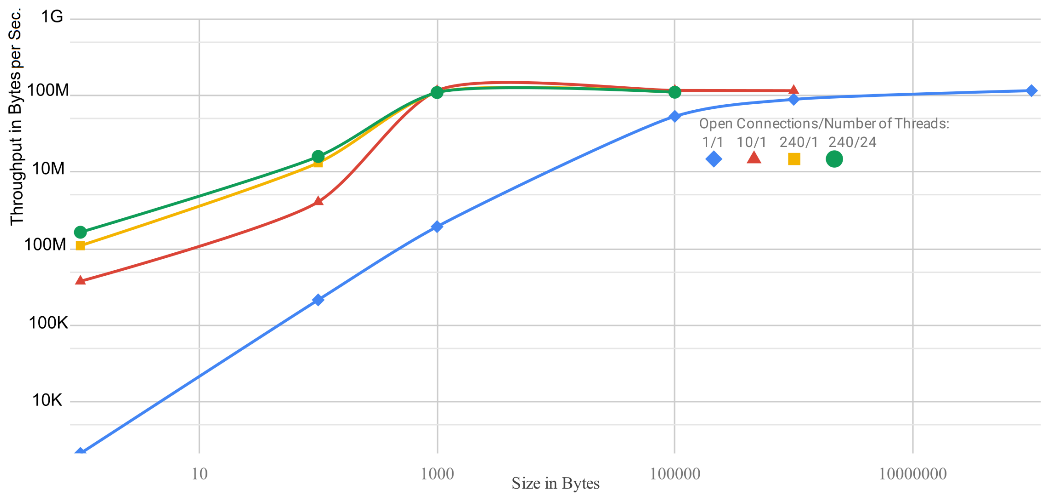


FIGURE 4.6: Throughput related to object size for different combinations of Open Connections/Threads.

4.3.5 Resource Usage Measurements

In addition to the latency diagrams and the findings acquired from them, another point to consider is the efficiency of the I/O itself; For this reason, we measure the memory and CPU usage needed to achieve a certain throughput. To accomplish this, the likwid-perfctr (Treibig, Hager, and Wellein, 2010) tool is used. It uses the Linux 'msr' module to access model-specific registers stored in /dev/cpu/*/msr (which contain hardware performance counters) and calculates performance metrics, FLOPS, bandwidth, etc., based on the event counts collected over the runtime of the application process.

The conducted experiment uses a setup similar to the one depicted by fig. 4.1; however, this time, we use a slightly different benchmarking tool, wrk (Glozer, 2020), since we want to stress test the system and hence there is no need to specify a maximum req/s rate. Figure 4.7 represents the modified setup.

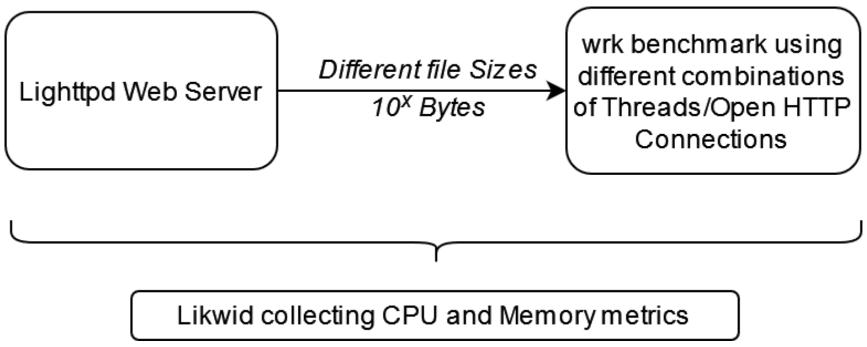

Lighttpd Web Server
Different file Sizes
10^X Bytes
wrk benchmark using different combinations of Threads/Open HTTP Connections
Likwid collecting CPU and Memory metrics

FIGURE 4.7: Simplified View of the Benchmark Setup using Likwid

Different files having a size of 10^x Bytes are created using random characters on the server side, then the benchmark is launched on the client side to request each of these files. Each iteration lasts **one** minute, and during this period, Likwid is recording the CPU performance counters, which are relevant in this scenario. The server application is pinned to one core using Likwid; the same is done on the client-side. To ensure that the process is run on the first physical core and not migrated between cores and avoid any overhead, the wrk tests are performed using only one thread.

CPU consumption is recorded, CPU_CLK_UNHALTED_CORE is the metric provided by Likwid that represents the number of clock ticks needed by the CPU to do some reasonable work. The instructions required to accomplish one request - by the server as well as by the client - seem to be constant for files having a size smaller than one KB and increase rapidly after this, as shown in fig. 4.8. Note that the server appears to be consuming more CPU cycles than the client to deliver a request, which might be because we use the Lighttpd web server without modifying the default configuration. Note also that over a specific file size limit and for a higher number of open connections, the number of timeouts and socket errors increase since we are approaching the maximum throughput that the system can achieve, causing the benchmark not to deliver any metrics for those values[4].

[4]The results of all the benchmarks conducted on the Mistral and WR Cluster, and the scripts needed to reproduce those benchmarks were published alongside the Paper entitled: Investigating the Overhead of the REST Protocol when Using Cloud Services for HPC Storage and are found online at `https://github.com/http-3/rest-overhead-paper`

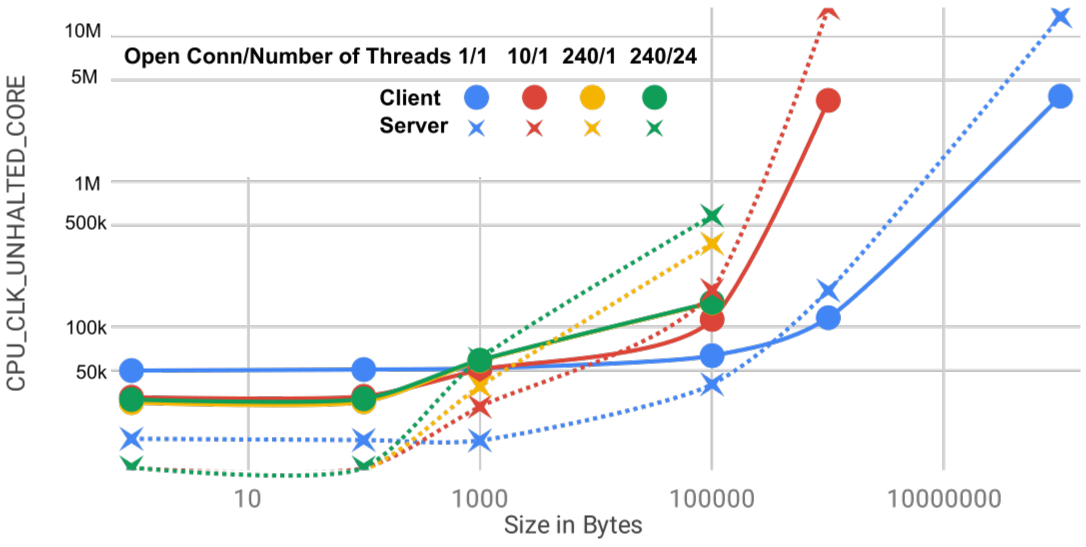


FIGURE 4.8: CPU usage for the client and server related to size, for different Open Connections/Threads combinations

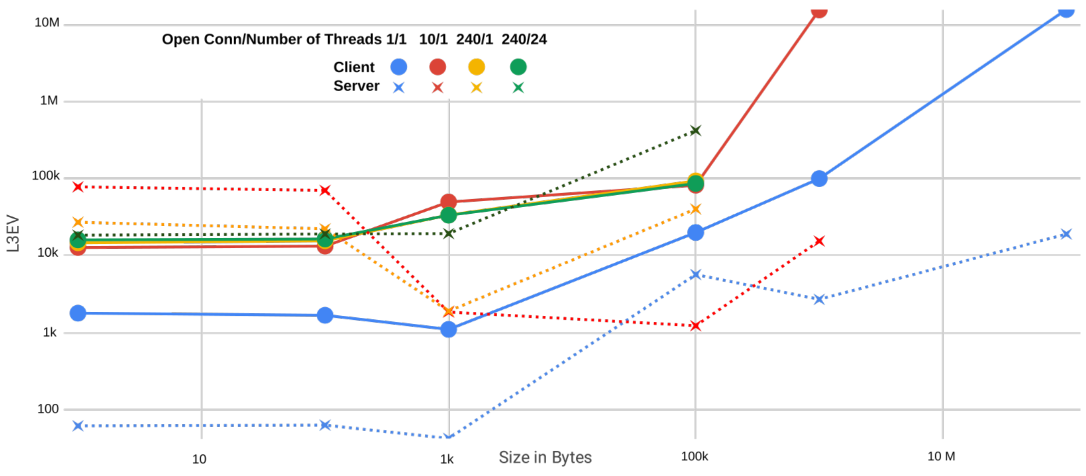


FIGURE 4.9: L3 evicted volume for the client and server related to size for different Open Connections/Threads combinations

Regarding memory utilization: Basically, when reading a file (represented by HTTP response), the client needs to store the data received in memory. If the file size exceeds the CPU cache size, we expect that data is evicted to main memory, which is measured in L3 cache evictions. This metric is recorded using Likwid and shown in fig. 4.9. Even for 100 MB files, we can see that only 10 MB of data is evicted on the client. There is no eviction on the server because it sends the data directly to the client. This is an indication that zero-copy (Tianhua et al., 2008) is in use on the client, and the network interface card offloads the processing of TCP/IP. This allows the network card to store the data directly into the target memory location. Typically, with zero-copy, the application requests the kernel to copy the data directly from a file descriptor to the socket bypassing the copy in user mode buffer and, therefore, reducing the number of context switches between kernel and user mode.

Furthermore, when data does not fit in the processor L3 caches (12 MB), the evicted data, i.e., the data passed to memory, increases significantly, causing a performance drop, curiously the rate of increase (slope) of the client evicted memory is more significant than the one on the server, leading us to another interesting conclusion, namely that **while most studies focused on optimizing the server-side, it might be the client-side that needs to be addressed**.

4.3.6 REST vs. MPI

As found in the previous tests, the available bandwidth plays an essential role in determining the achieved latency and throughput. The following tests are conducted on Mistral where Infiniband (Association, 2020) is available.

Our next step is to compare the REST protocol with an established data transfer approach in the HPC world, namely the Message Passing Interface MPI (The MPI Forum, 1993). Although the MPI programming interface has been standardized, different library implementations exist; we will be only considering and using the Open MPI implementation (Graham, Woodall, and Squyres, 2005) for the rest of this work.

To achieve this, we launch the same tools used above (likwid+lighttpd) on one node and (likwid+wrk) on another node while varying the file size in a power of 2 and recording the different metrics; the transfer uses the TCP protocol and takes place over the Infiniband interface.

Then we launch the OSU Micro Benchmark (Liu et al., 2004) alongside with likwid on two nodes using the same file sizes and record the same metrics. The OSU tests are executed over Infiniband, the first time using RDMA (Remote Direct Memory Access), and the second time utilizing TCP.

The OSU Benchmark offers two types of tests:

- osu_get_latency - Latency test, where the latency represents the time taken to transfer a message of a specific size from one MPI rank to another. Separate buffers are used for sending and receiving but stay the same during each iteration. The equation used to calculate the throughput is illustrated in fig. 4.10.

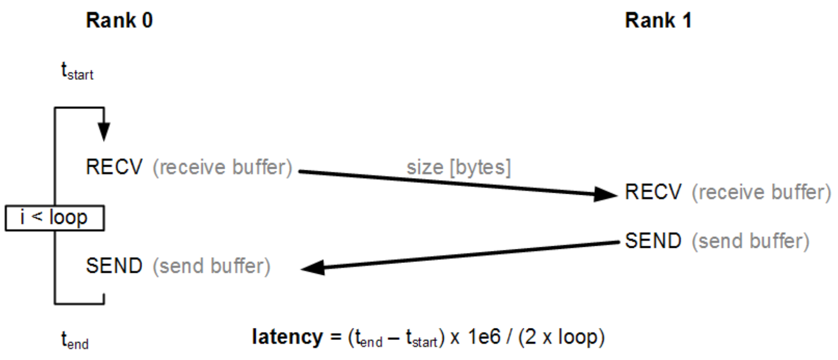


FIGURE 4.10: Visualisation of the OSU Latency Benchmark (Wittman, 2014)

- osu_get_bw - Throughput test, used to test the unidirectional throughput from one MPI Rank to another: several MPI_Isends are initiated, followed by an MPI_Waitall. The receiving side uses matching MPI_Irecvs with MPI_Waitall, and one iteration ends when the sending side receives all messages acknowledged. The window_size represents the number of started MPI_Isends. The equation used to calculate the throughput is illustrated in fig. 4.11. The collected metrics for this test are obtained using Likwid and are compared to those obtained from the REST benchmark.

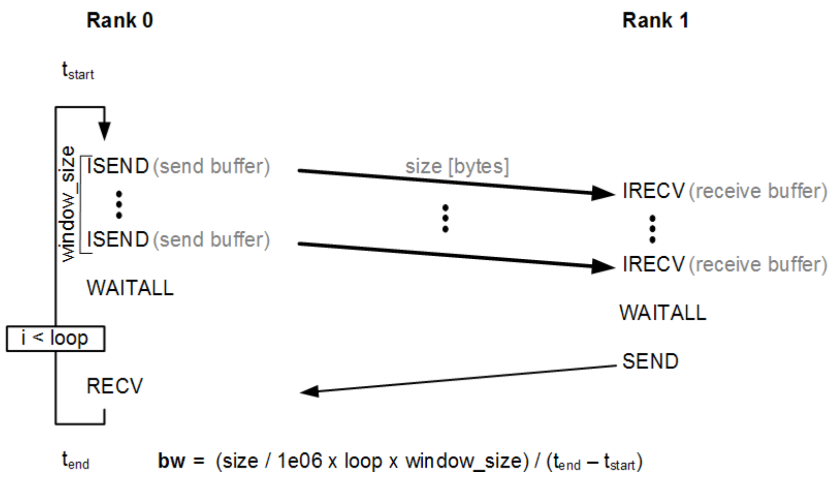


FIGURE 4.11: Visualisation of the OSU Throughput Benchmark (Wittman, 2014)

The obtained results are used to plot fig. 4.12 and fig. 4.13. Figure 4.12 illustrates the latency results for the REST benchmark over TCP over InfiniBand and for the osu_get_latency MPI benchmark over TCP over InfiniBand, and over RDMA over InfiniBand. Figure 4.13 displays the throughput results for the REST benchmark over TCP over InfiniBand and for the osu_get_bw MPI benchmark over TCP over InfiniBand, and over RDMA over InfiniBand as well.

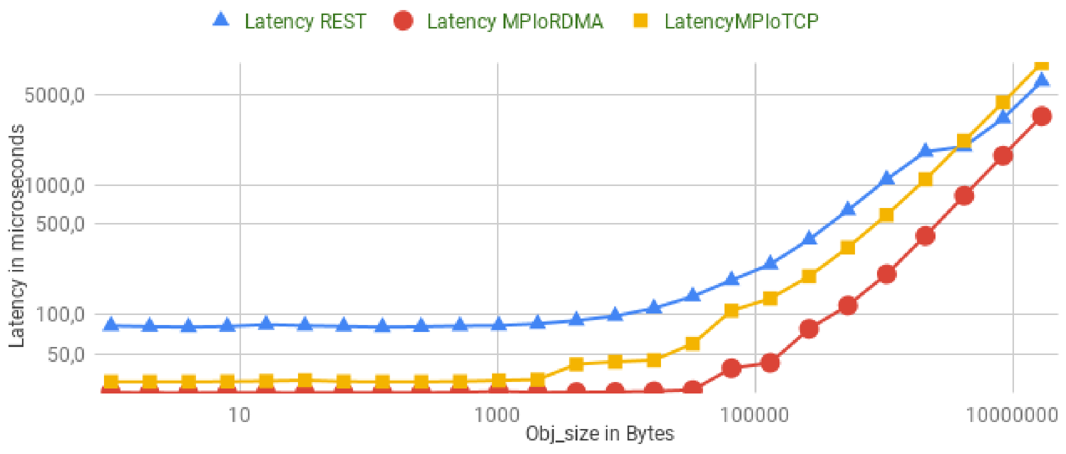


FIGURE 4.12: Latency results for the different protocols related to the file size

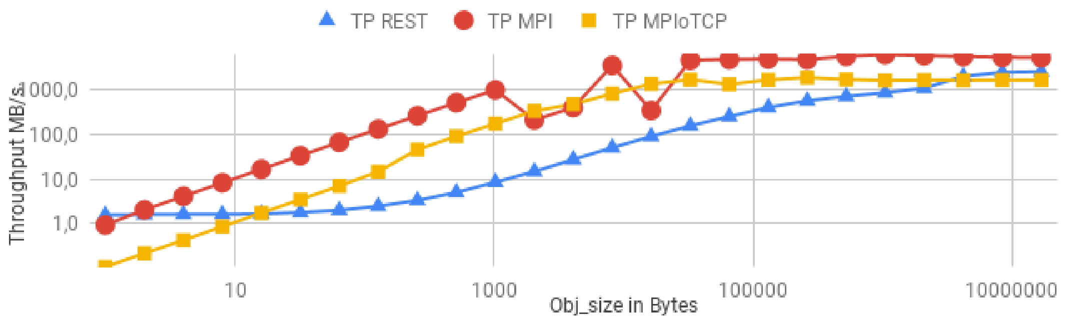


FIGURE 4.13: Throughput results for the different protocols related to the file size

Our observations and interpretations are as follows:

- For small object sizes, the latency of Rest is obviously higher than the one of MPI. As already mentioned in our latency tests, this is due to the HTTP overhead.
- The throughput achieved using MPI is better than the one using REST; however, when comparing MPI and REST both over TCP, we notice that this is not the case especially for very small and for large files as well, which leads to the conclusion that the overhead due to the TCP stack is the main factor slowing down REST and thus an object storage implementation.
- The performance dip seen in the red line for a file size of above 1 KB is due to the MPI implementation that uses a combination of protocols for the same MPI routine (Denis and Trahay, 2016), namely the use of the eager protocol for small messages, and rendezvous protocol for larger messages.

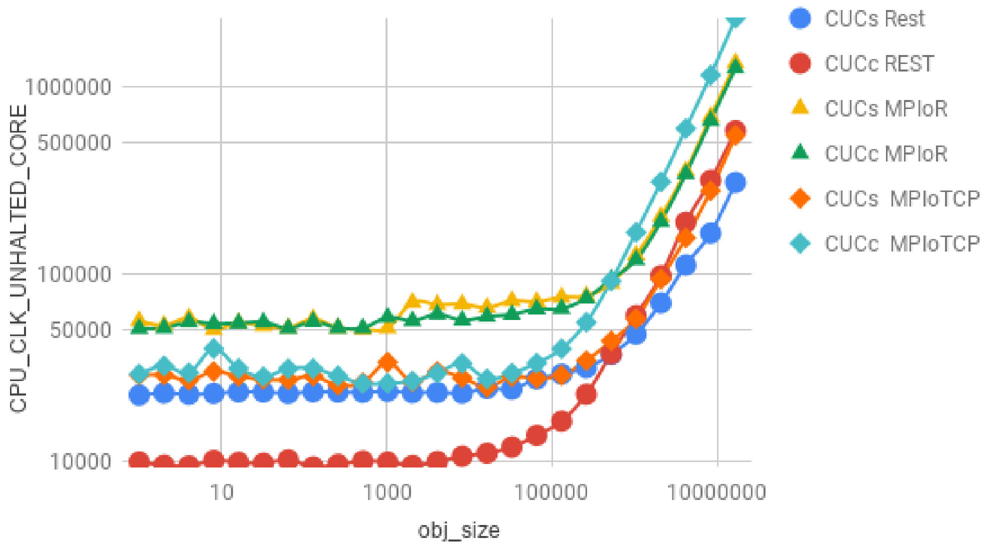


FIGURE 4.14: CPU Unhalted Cycles per request, on the server and client, for each protocol

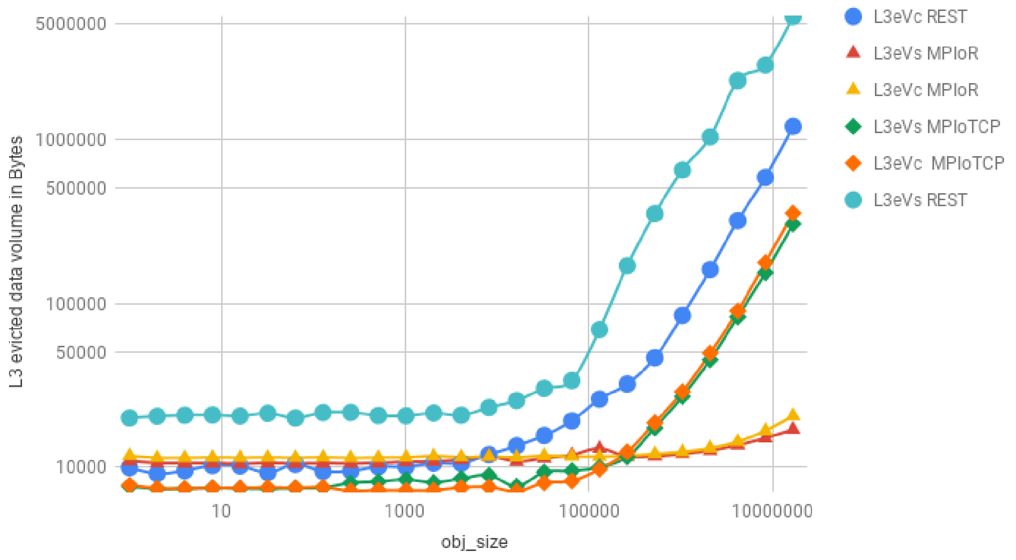


FIGURE 4.15: L3 evicted per request, on the server and client, for each protocol

- Another particular finding depicted by fig. 4.14 is that the number of CPU cycles needed for the sender to push the data when using MPI is higher than when using REST; this becomes more visible for file sizes above 100 KB.
- Figure 4.15 shows that, as expected, the evicted data volume stays constant in the case of MPI over RDMAoIB because of the direct data transfer from the server main-memory to the client main-memory. Furthermore, the L3-evicted memory for both REST and MPI over TCPoIB is constant for files smaller than 100 KB. Still, it increases exponentially afterwards, presumably, because parts of the protocol such as network packets re/assembly are controlled by the kernel and not the network interface.

4.3.7 HTTP Size Overhead

In addition to the protocol overhead and packet fragmentation introduced by the communication protocol in use, TCP in our case, we are also interested in the overhead due to the use of HTTP.

To calculate the overhead per request, in this case for HTTP 1.1, we have the amount of bytes_read by the HTTP parser in wrk, and since we know the number of requests achieved, we can assume that:

$$http\_overhead\_per\_request = \frac{bytes\_read}{Nreq} - objsize$$

The overhead is about 233 bytes for every request, mainly due to the uncompressed, literally redundant HTTP response headers, which can constitute a significant portion of the HTTP traffic, specifically for large numbers of HTTP requests for files sizes smaller than 1 KB.

4.4 Evaluation of the Performance Model

To validate the predictive model defined in eq. (4.9), we use the values reported by the REST latency Benchmark on Mistral in section 4.3.6; the hardware-specific parameters are calculated as follows:

Data between sockets and memory is shipped via a 9.6 GT/s QPI interface (Intel, 2014). According to the Intel QPI specification (Intel, 2009), 16 bits of data are transferred per cycle; thus, the uni-directional speed is 19,6 GB/s. The communication protocol has an overhead of roughly 11% ,therefore, mem_tp = 17 MB/s.

The compute nodes of Mistral are integrated into one FDR InfiniBand fabric; the measured bandwidth between two arbitrary compute nodes is 5.9 GByte/s, as such net_tp = 5,9 GByte/s, rtt measured using the tool qperf[5] and found = 0.06ms and mtu = 65520 Bytes.

We only need to get the values of the coefficients β_i in eq. (4.9). This is done by using a regression analysis tool, in this case, the one provided by Excel: the obtained R square and F values are examined, for each iteration, to check the fitness and the statistical significance of our model, respectively. Finally, we calculate the predicted values and compare them to the ones obtained in the benchmark by determining the error rate using eq. (4.10).

$$error\% = (t\_req - t\_req\_calcul) \cdot 100 / t\_req \tag{4.10}$$

In case of RESToTCPoIB, we find that:

$\alpha = 1$

$\beta_1 = \beta_3 = \beta_4 \sim 1$

$\beta_2 = 6$ and $\beta_5 = 3/2$

The results are shown in appendix A within table A.1, the deviation (error rate) between the estimated value and the benchmark results is primarily below 10 percent, and indeed in the range of 1 percent for small and large file sizes. Equation (4.9) yields:

$$t(request) = rtt + \frac{CUCs}{Rs} + 6 \cdot \frac{L3EVs}{mem\_tp} + \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + \frac{3}{2} \cdot \frac{Obj\_size}{net\_tp} \tag{4.11}$$

In case of MPIoTCP, we obtain:

$\alpha = 0.1$

$\beta_1 = \beta_2 = \beta_3 = \beta_4 \sim 1$, and $\beta_5 = 2.7$

As shown in appendix A in table A.2, the deviation (error rate) between the estimated value and the calculated one is primarily below 20 percent, and less than 5 percent for small and large file sizes. Equation (4.9) yields :

$$t(request) = 0.1 \cdot rtt + \frac{CUCs}{Rs} + \frac{L3EVs}{mem\_tp} + \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + 2.7 \cdot \frac{Obj\_size}{net\_tp} \tag{4.12}$$

In case of MPIoRDMA, we obtain:

$\alpha = 0$

$\beta_1 = \beta_3 = 1/2$ and $\beta_2 = \beta_4 = \beta_5 \sim 1$

[5]https://linux.die.net/man/1/qperf

As shown in appendix A in table A.3, the deviation (error rate) between the estimated value and the calculated one is primarily below 10 percent, and less than 8 percent for small and large file sizes. Equation (4.9) yields :

$$t(request) = \frac{1}{2} \cdot \frac{CUCs}{Rs} + \frac{L3EVs}{mem\_tp} + \frac{1}{2} \cdot \frac{CUCc}{Rc} + \frac{L3EVc}{mem\_tp} + \frac{Obj\_size}{net\_tp} \tag{4.13}$$

Section 4.4 summarizes the obtained model coefficients for the different protocols.

Protocol	α	β_1	β_2	β_3	β_4	β_5
RESToTCPoIB	1	1	6	1	1	3/2
MPIoTCPoIB	0.1	1	1	1	1	2.7
MPIoRDMAoIB	0	1/2	1	1/2	1	1

TABLE 4.1: Model coefficients for the different protocols

We can infer some general behavior by investigating the model terms and thus verify our expectations. The latency for MPIoRDMA is expected to be lower than the other transfer methods, and this is why α is close to 0 for this model. If β_5 is above 1, it is an indicator that we cannot achieve full network throughput. REST and MPIoTCP show otherwise similar performance characteristics. At the same time, the MPIoRDMA model uses approximately half the CUC, which actually means it needs twice as many CPU Cycles compared to the TCP models - which may be due to busy waiting. These assumptions can be verified by looking at fig. 4.14 and fig. 4.15.

We conclude that while TCP proved itself for end-to-end communications over long distances, it is less suitable for data center networking, mainly because of its processing overhead (CPU and memory resources consumption), hence degrading the aspired performance. On the other side, CPU and memory consumption for the REST over TCP Model remains adequate compared to MPI over TCP and MPI over RDMA.

4.4.1 Comparison: HTTP1.1 vs HTTP2 vs HTTP3

In this section, we compare the performance of the different versions of the HTTP protocol.

Figure 4.16 shows the benchmark used to achieve this purpose. It is similar to fig. 4.1; however, in this case, the webserver should be able to deliver the three different protocols, and the benchmark client should be able to communicate using the various HTTP protocols. Therefore, openlitespeed (OpenLiteSpeed, 2020) is used as a webserver alongside the suitable benchmark tool, which is h2load (h2load, 2020).

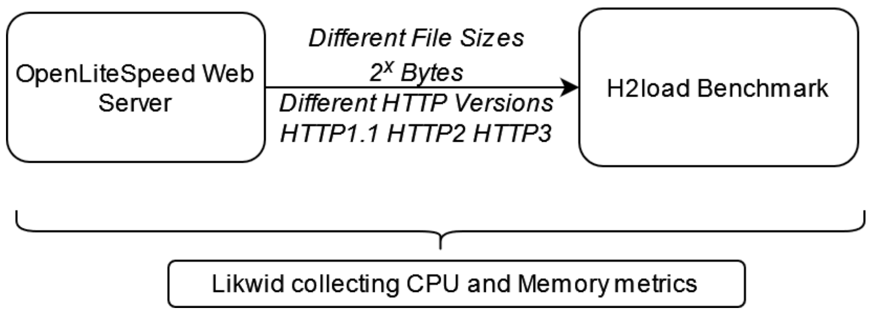


FIGURE 4.16: Simplified View of the Benchmark used to test the different versions of HTTP

To note that we test here the ngtcp2 (ngtcp2, 2020) implementation of HTTP3 because it is TLS library independent, not like other HTTP3 implementations, for example, quiche (Cloudflare, 2020) which requires the boringssl library explicitly. Since, at the time of writing, the official OpenSSL Team does not support QUIC (OpenSSL, 2020) we use a patched version of OpenSSL provided by the ngtcp2 team.

Since HTTP3 did not achieve the maturity phase yet, we are using the protocols as they are defined in the 27th Draft by the IETF QUIC Working group (IETF, 2020).

The latency and throughput results of the tests on the WR cluster are shown in fig. 4.17 and fig. 4.18 respectively:

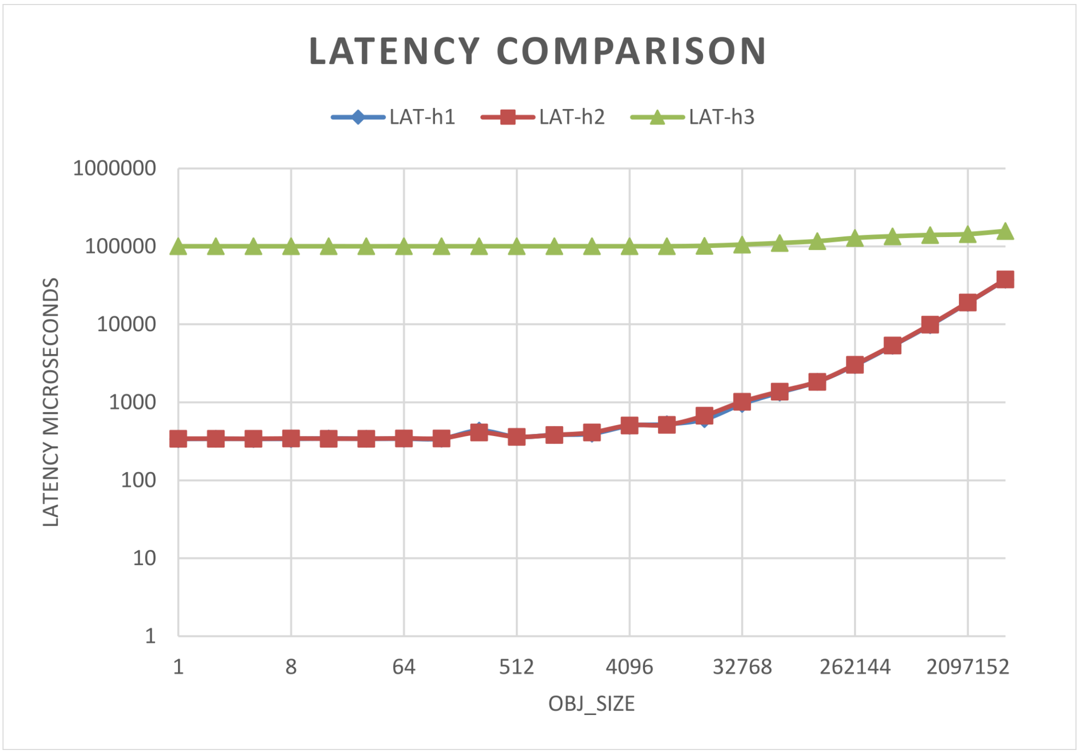


FIGURE 4.17: Latency results for the different protocols on the WR Cluster

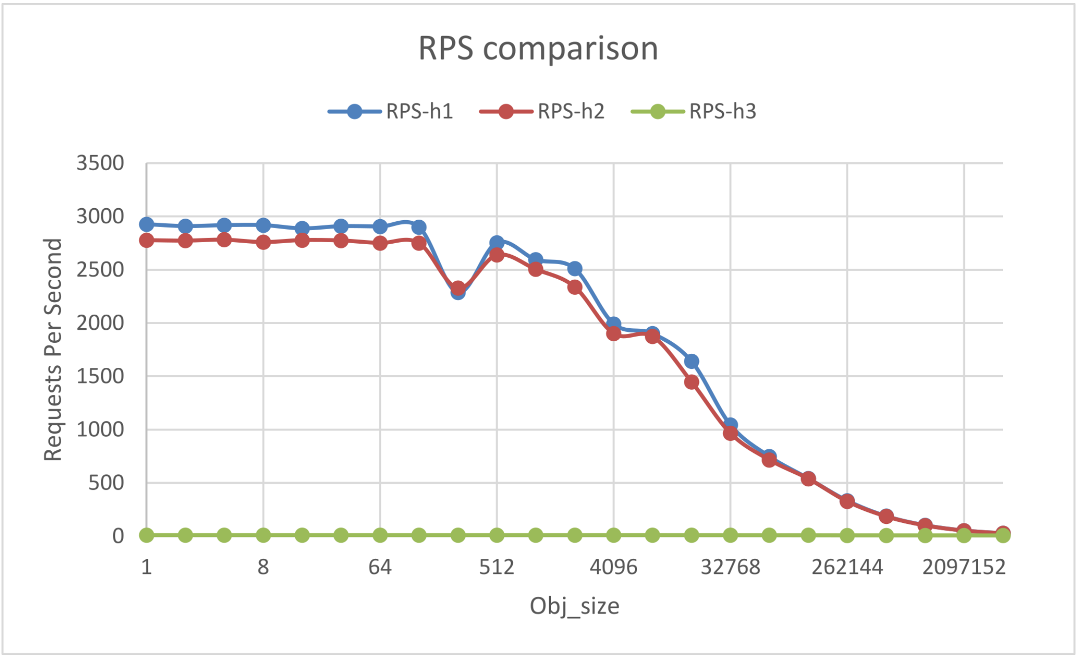
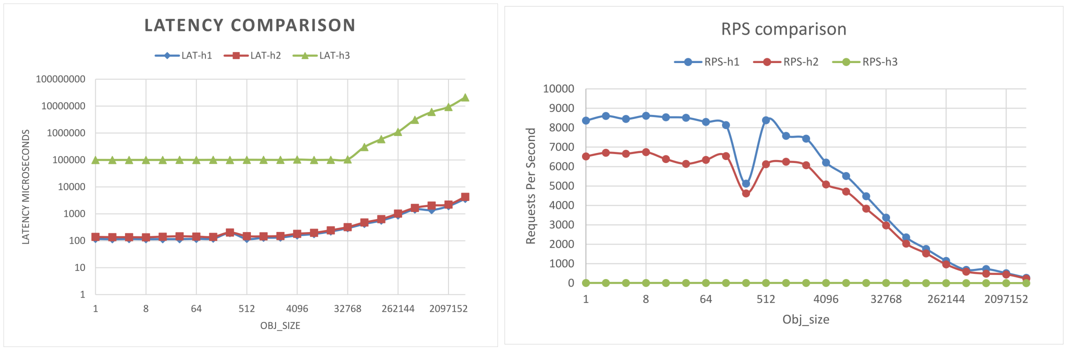


FIGURE 4.18: Throughput results for the different protocols on the WR Cluster

As seen, the latency and throughput results for both HTTP1.1 and HTTP2.2 are very identical in contrast to HTTP3, which appears to be performing poorly. The same tests were also conducted on Mistral over InfiniBand, and similar results were obtained, as illustrated in fig. 4.19a and fig. 4.19b

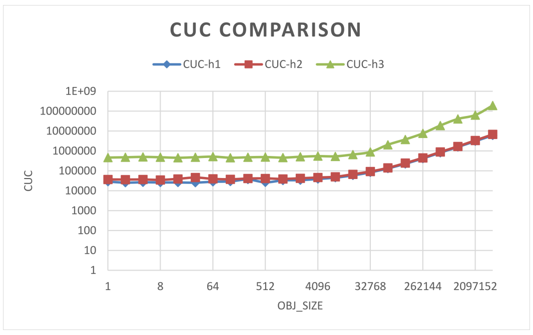


(A) Latency Results (B) Throughput Results

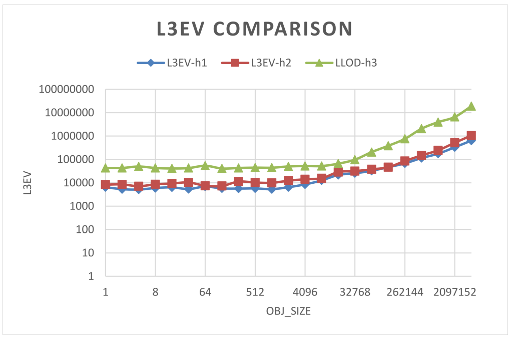
FIGURE 4.19: Performance Comparison for the different protocols on Mistral

Although we expect HTTP2 and HTTP3 to perform better than HTTP 1.1, this is not the case. A closer look at the evolution of the parameters defined in our model reveals the cause:

- Figure 4.20a exhibits a ten-fold increase in CPU Cycles on the client-side when using the HTTP3 protocol in comparison to HTTP1.1 and HTTP2.
- Figure 4.20b also indicates a ten-fold growth in the L3 eviction rate, hence an increase in main memory usage, on the client-side when utilizing the HTTP3 protocol in comparison to HTTP1.1 and HTTP2.



(A) Client CPU Cycles Usage



(B) L3 Eviction Rate

FIGURE 4.20: Resource Usage Comparison for the different protocols on Mistral

Despite the apparent traffic saving of HTTP2, its memory and CPU consumption are relatively equivalent to HTTP1.1, thus based on our model from section 4.2.1, the latency from both protocols is similar, as we can see in fig. 4.19a . On the other hand, the chosen HTTP3 implementation is circa ten times CPU and memory-consuming compared to the earlier versions, which indicates an implementation issue.

4.5 Summary

This chapter provides a first assessment of using REST as a storage protocol in an HPC environment.

A performance model for the relevant HTTP Get/Put operation based on hardware counters is provided and experimentally validated. Our results demonstrate that a correctly configured REST implementation can provide high performance and match HPC-specific implementation of MPI in terms of throughput for most file sizes and in terms of latency for file sizes above 1 MB.

By considering the CPU and memory cost introduced by the data movement, the developed model covered well the general behavior of the different protocols and confirmed the expected behavior.

The new techniques introduced in the more recent versions of HTTP (the use of a small number of connections, multiplexing the HTTP datagram, compressing the header, and allowing the server to "push" data pro-actively to the client while eventually using UDP to accomplish these) bear the potential to improve performance and, thus, provide a perspective for using cloud storage within HPC environments. However, in this evaluation, they could not show their benefit.

The study presented in this chapter paves the way for chapter 5, where we assess in more detail the performance of different object storage implementations on HPC and in the cloud. These implementations have one thing in common, they all offer an S3 compatible API, allowing us to use the same benchmark to provide a fair and comprehensive comparison. This will eventually help us achieve one of our research goals, i.e., examining if an object storage implementation on top of REST is a performant and efficient alternative to common HPC storage in an actual HPC scenario.

Chapter 5

S3 Performance Analysis for HPC Workloads

In chapter 4, we found that the performance overhead due to the mere use of REST/HTTP is to some extent comparable to the one introduced by HPC native protocols like MPI. This chapter goes a step further by assessing the performance of different cloud storage implementations. The Simple Storage Service S3 has emerged as the de-facto storage API for object storage in the cloud and represents, as such, a standard interface to access these implementations. We seek to check if the S3 API is already a viable alternative for HPC access patterns in terms of performance or if further performance advancements are necessary. For this purpose, we extend two typical HPC I/O benchmarks — the IO500 and MD-Workbench — to quantify the performance of the S3 API. We perform the analysis on the Mistral supercomputer by launching the enhanced benchmarks against different S3 implementations: on-premises (Swift, MinIO) and in the cloud (Google, IBM...). Section 5.2 describes the test scenarios and defines the relevant metrics that will be addressed using our benchmarks. Section 5.3 describes the experimental procedure, the used systems, and the methodology of the evaluation conducted in this work. Section 5.3.4 analyzes the obtained latency results. Section 5.4 summarises our findings.

Most of the content of this chapter has been published under (Gadban and Kunkel, 2021)

5.1 Foreword

With the increased prevalence of cloud computing and the increased use of the Infrastructure as a Service (IaaS), various APIs are provided to access storage. The Amazon Simple Storage Service (S3) (AWS, 2020) managed to be the most widely adopted cloud storage API in the cloud, and it is being increasingly used for HPC workloads (Greguska, 2018; Gutierrez and Jesus, 2020). Many cloud storage providers, like IBM, Google, and Wasabi, offer S3 compatible storage, and a large number of Scale-Out-File Systems like Ceph (Weil et al., 2006), OpenStack Swift (OpenStack-Foundation, n.d.) and Minio (MinIO-Inc., n.d.) offer a REST gateway, largely compatible with the S3 interface. HPC applications often use a higher-level I/O library such as NetCDF (Rew and Davis, 1990b) or ADIOS (Lofstead et al., 2008) or still the low-level POSIX API. Under the hood, for the interaction with the storage system, MPI-IO and POSIX are still widely used, while other object storage APIs such as the native DAOS (Lofstead et al., 2016) API[1] are still emerging.

As mentioned in chapter 2, if the performance characteristics of S3 are promising, it could be used as an alternative backend for HPC applications. This interoperability would foster convergence between HPC and cloud and eventually lead to

[1] https://docs.daos.io/v2.0/overview/architecture

consistent data access and exchange between HPC applications across data centers and the cloud.

After the release of the Amazon S3 service in 2006, many works were published to assess the performance of this offering. Some of them (Garfinkel, 2007; Palankar et al., 2008) focused only on the download performance of Amazon S3, most of them (Garfinkel, 2007; Palankar et al., 2008; Bessani et al., 2013; Arsuaga-Ríos et al., 2015; Sadooghi et al., 2015) never published or described the used benchmarks, others (Garfinkel, 2007; Palankar et al., 2008; Inc, n.d.) are not able to assess S3 compatible storage. The Perfkit benchmarker from Google was used in (Bjornson, 2015) to compare the download performance of AWS S3 in comparison with Google Cloud Storage (GCS[2]) and Microsoft Azure Storage. However, since the tests were accomplished in the cloud, the obtained results depend heavily on the VM machine type in use, hence, on the network limitation enforced by the cloud provider; Adding to the confusion, and in contrast, (Sadooghi et al., 2015) found in their tests, which were also run in the AWS cloud on different EC2 machines against the Amazon S3 implementation, that "there is not much difference between the maximum read/write throughput across instances."

As such, the lack of published tools that cover HPC workloads pushed us to enhance two benchmarks already used for HPC procurements, namely IO500[3] and MD-Workbench (Kunkel and Markomanolis, 2018), by developing a module capable of assessing the performance of S3 compatible storage. We will explore our approach in more depth in the upcoming section.

5.2 Methodology

We aim to analyze the performance of the S3 interface of different vendors in an HPC environment and eventually assess the performance potential of the S3 API for HPC workloads. To achieve our purpose, a five steps procedure is implemented by:

- Identifying suitable benchmarks.
- Modifying the benchmark to support S3.
- Running the enhanced benchmarks against the S3 interface presented by various on-premises and in-cloud object storage implementations.
- Determining a measurement protocol that allows identifying the main factors affecting the performance of S3 for HPC workloads.
- Providing alternative implementations for S3 to estimate the best performance.

Figure 5.1 represents a high-level overview of this approach, depicting the different object storage stacks being considered the various factors of comparison alongside the different benchmarks in use.

[2]Google Cloud Storage
[3]https://io500.org

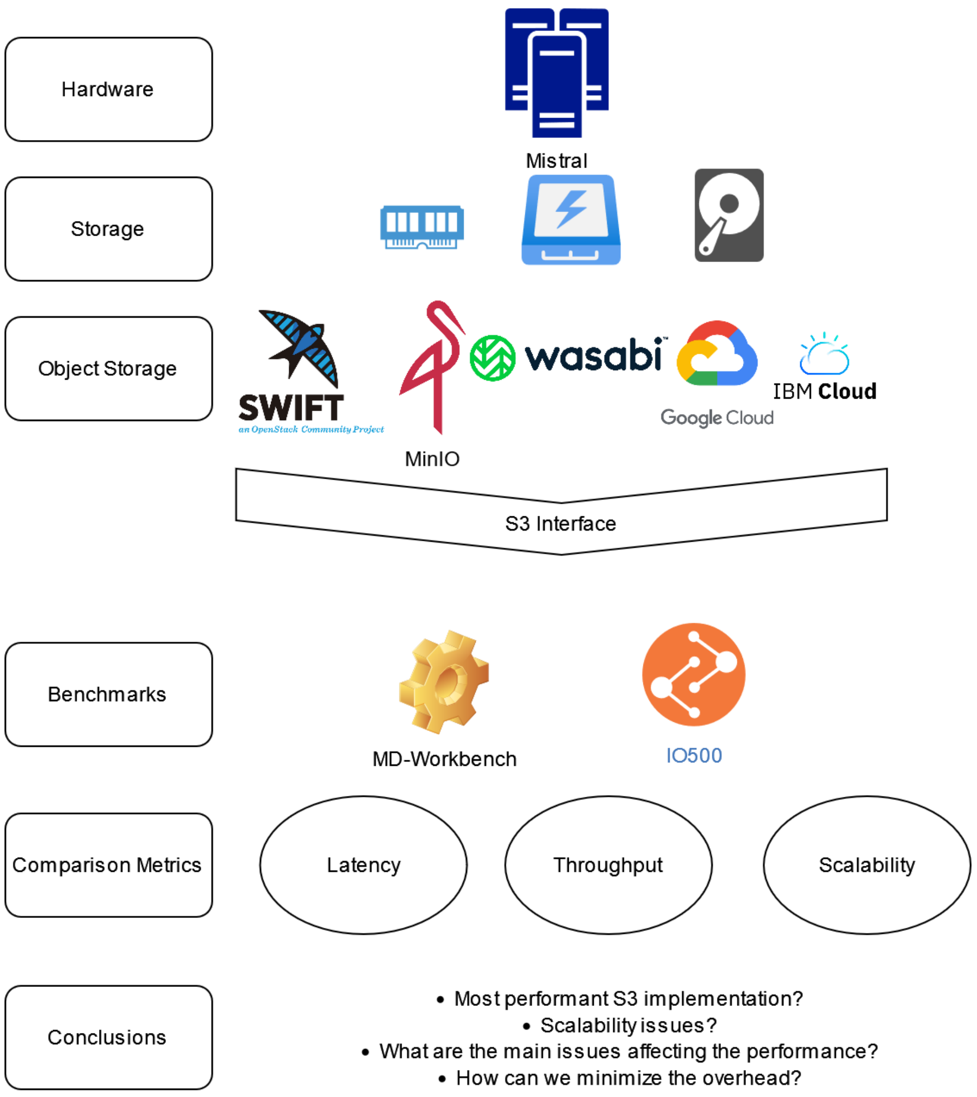


FIGURE 5.1: High Level Overview of the S3 Performance Analysis Methodology

5.2.1 Benchmarks

Two HPC benchmarks, the IO500 and MD-Workbench, are extended in order to analyze the potential peak performance of the S3 API on top of the existing HPC storage infrastructure.

- The IO500 benchmark consists of multiple subcomponents: bandwidth subcomponents, metadata subcomponents, and namespace searching subcomponents. The bandwidth subcomponents and metadata components use the IOR and mdtest benchmarks, respectively. These benchmarks simulate a variety of typical HPC workloads, including the bulk creation of output files from a parallel application, intensive I/O operations on a single file, and the post-processing of a subset of files. The IO500 uses the IOR/MDTest benchmark under the hood, which comes with a legacy backend for S3 using the outdated aws4c library (Korolev, n.d.) that stores all data in a single bucket; as such, a file is one object which is assembled during write in the parallel job using multipart messages – most S3 implementations do not support this.

The IO500 (Kunkel, Lofstead, and Bent, 2017) uses IOR and MDTest in an "easy" and "hard" setup, hence performs various workloads and delivers a single score for comparison; the different access patterns are covered in different phases:

  - IOEasy simulates applications with well-optimized I/O patterns.
  - IOHard simulates applications that utilize segmented input to a shared file.
  - MDEasy simulates metadata access patterns on small objects.
  - MDHard simulates accessing small files (3901 bytes) in a shared bucket.

We justify the suitability of these phases as follows: B. Welch and G. Noer (Welch and Noer, 2013) found that, inside HPC, between 25% and 90% of all files are 64 KBytes or less in size, as such a typical study of the performance of object storage inside HPC should also address this range, rather than only focusing on large sizes, which are expected to deliver better performance and only be limited by the network bandwidth (Bjornson, 2015; Gadban, Kunkel, and Ludwig, 2020).

This is why, using the IOR benchmark, we highlight this range as shown in fig. 5.3 and display the performance for size up to 128 MiB in section 5.3.2 and section 5.3.3. Large file sizes are also addressed since the performed IO500 benchmarks operate on 2 MiB accesses, creating large aggregated file sizes during 300 seconds run.

- The MD-Workbench (Kunkel and Markomanolis, 2018) is used to explore interactive operations on files: it simulates concurrent access to typically small objects and reports throughput and latency statistics, including the timing of individual I/O operations. The obtained latency results are used to plot a density graph of the individually timed operations and help us identify which I/O operations are the least performant.

  The benchmark executes three phases: pre-creation, benchmark, and cleanup.

  - The pre-creation phase setups the working set while the cleanup phase removes it.
  - A pre-created environment that is not cleaned can be reused for subsequent benchmarks to speed up regression testing, i.e., constant monitoring of performance on a production system.
  - The working set is kept constant during the benchmark run: in each iteration, a process produces one new object and then consumes a previously created object in FIFO order.

5.2.2 Modifications of benchmarks

For IO500, an optimistic S3 interface backend using the libS3 client library is implemented for IOR in the sense that it stores each fragment as one independent object. As such, it is expected to generate the best performance for many workloads.

For identifying bottlenecks, it supports two modes:

- single bucket mode: created files and directories results in one empty dummy object (indicating that a file exists), every read/write access happens with exactly one object (filename contains the object name + size/offset tuple); deletion traverse the prefix and removes all the objects with the same prefix recursively.
- one bucket per file mode: for each file, a bucket is created. Every read/write access happens with exactly one object (object name contains the filename + size/offset tuple); deletion removes the bucket with all contained objects.

Consequently, the libs3 implementation gives us the flexibility to test some optimistic performance numbers. The S3 interface does not support the "find" phase of IO500, which we, therefore, exclude from the results.

MD-Workbench recognizes datasets and objects and also offers two modes:

- one bucket, the D datasets are prefixed by the process rank.
- one bucket per dataset.

In both modes, objects are atomically accessed, fitting the S3 API directly.

The libS3 used in IO500 is the latest one which only supports AWS signatures v4 (bji, n.d.) while the current release[4] of MD-Workbench supports an older version of libs3, which uses the AWS signature v2. As such, it is ideal for the benchmarking of some S3 compatible systems that only support the v2 signature, like the one found at DKRZ[5].

Listing 5.1 presents the different arguments used by the introduced IOR S3 interface.

LISTING 5.1: The different arguments for the introduced IOR S3 interface

```
Module S3-libs3

Flags
  --S3-libs3.bucket-per-file     Use one bucket to map one
     file/directory, otherwise one bucket is used to store
     all dirs/files.
  --S3-libs3.dont-suffix-bucket By default a hash will be
     added to the bucket name to increase uniqueness, this
     disables the option.
  --S3-libs3.s3-compatible       to be selected when using S3
     compatible storage
  --S3-libs3.use-ssl             used to specify that SSL is
     needed for the connection

Optional arguments
  --S3-libs3.bucket-name-prefix=iorThe prefix of the bucket(s
     ).
  --S3-libs3.host=STRING         The host optionally followed
     by:port.
  --S3-libs3.secret-key=STRING   The secret key.
  --S3-libs3.access-key=STRING   The access key.
```

[4] https://github.com/JulianKunkel/md-workbench/tree/0a26b061fae43fa0b72ac6d6b7ca2ae4621c54c9
[5] The German Climate Computing Center

```
--S3-libs3.region=STRING        The region used for the
    authorization signature.
--S3-libs3.location=STRING      The bucket geographic
    location.
```

5.2.3 Measurement protocol

We measure the performance on a single node and then on multiple nodes while varying the size of the object and the number of processes/threads per node. To assess the performance of the different modes, we establish performance baselines by measuring performance for the network, REST, and the Lustre file system. Then the throughput is computed (in terms of MiB/s and Operations/s) and compared to the available network bandwidth of the nodes.

5.3 Experiments

5.3.1 Test System

The tests are performed on the supercomputer Mistral (DKRZ, 2020), the HPC system for earth-system research operated at the German Climate Computing Center (DKRZ). It offers 3000 compute nodes, each equipped with an FDR Infiniband interconnect and a Lustre storage system with 54 PByte capacity distributed across two file systems. The system provides two 10 GBit/s Internet uplinks to the German research network (DFN); however only accessible on a subset of nodes.

5.3.2 MinIO Benchmarks in HPC

To create a reference number for the performance of S3 and explore the possible ways to optimize performance, we first use the MinIO server (release: 2020-08-18T19-41) to accomplish our tests using the modified benchmarks inside our HPC environment.

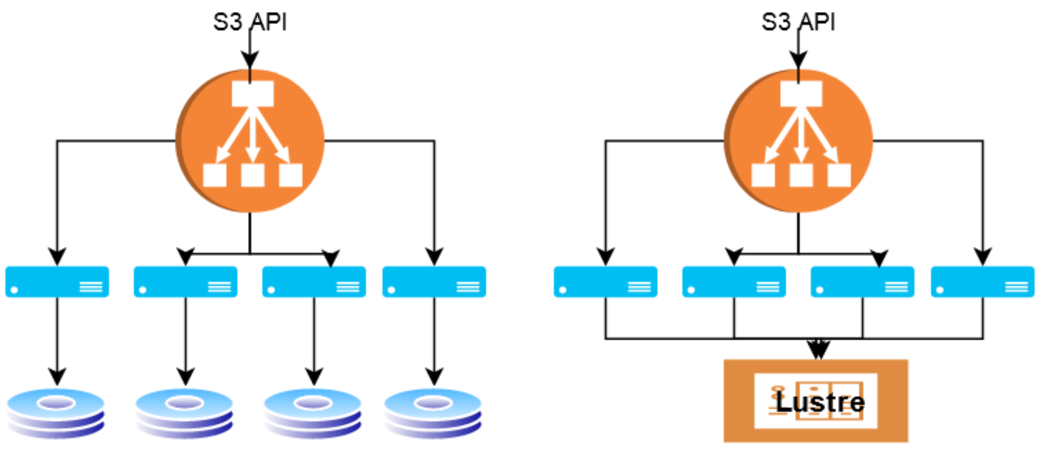
MinIO Deployment

MinIO supports the following modes of operation:

- Standalone (*sa*): runs one MinIO server on one node with a single storage device. We test configurations from tmpfs (in-memory fs/shm) and the local ext4 file system.
- Distributed servers (*srv*): runs on multiple nodes, object data and parity are striped across all disks in all nodes. The data is protected using object-level erasure coding and bitrot. Objects are accessible from any minio server node. In our setup, each server uses the local ext4 file system.

  Figure 5.2a illustrates the deployment.
- Gateway (*gw*): adds S3 compatibility to an existing shared storage. On Mistral, we use the Lustre distributed file system as the backend file system as seen in Figure 5.2b

Alongside these three modes, we introduce two modes by inserting the Nginx (Sysoev, n.d.) (v1.18.0) load balancer in front of the distributed and gateway configurations, and we refer to these setups as *srv-lb* and *gw-lb* respectively. Both variants can utilize a cache on the Nginx load balancer (*-cache*).



(A) Distributed Servers mode (B) Gateway mode

FIGURE 5.2: Different MinIO Modes

Single Client

The first tests are performed using IOR (LLNL, n.d.) directly. Figure 5.3 and Figure 5.4 show the performance on 1 node for a variable object size for the different MinIO modes. We can notice that the standalone mode shows the best performance. Gateway mode is effective for reads but slow for writes. Write performance is 1/3rd of the read performance. Compared to the Infiniband network throughput (about 6 GiB/s), only 7.5% and 2.5% of this performance can be obtained for reading and writing, respectively. Adding a load balancer has minimal influences on throughput in this setting, except when activating the caching mechanism, which significantly impacts the read throughput.

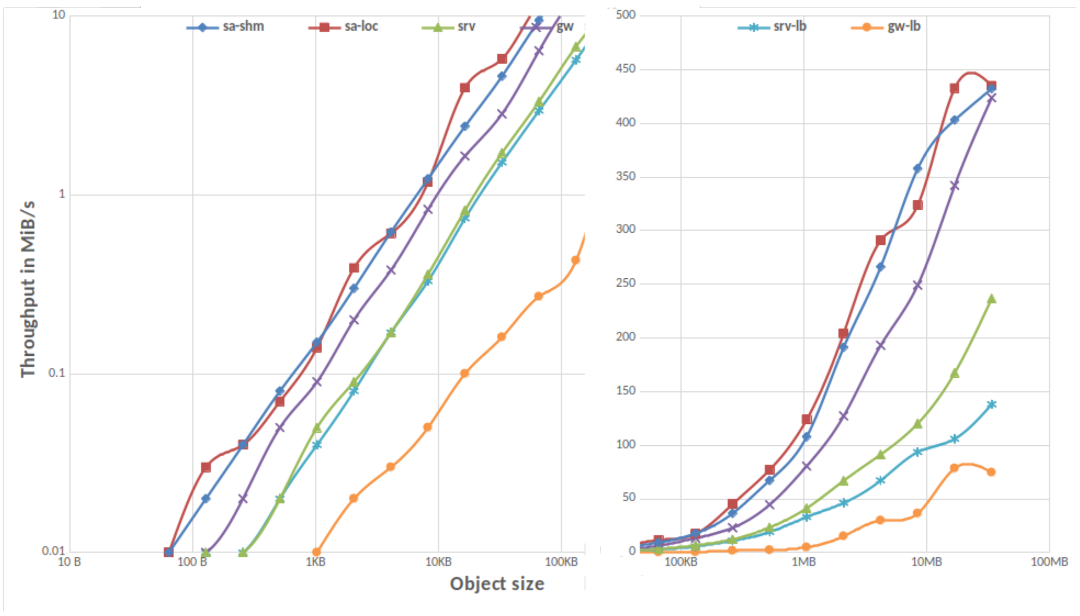


FIGURE 5.3: Read Throughput for MinIO modes for 1 node and 1 PPN

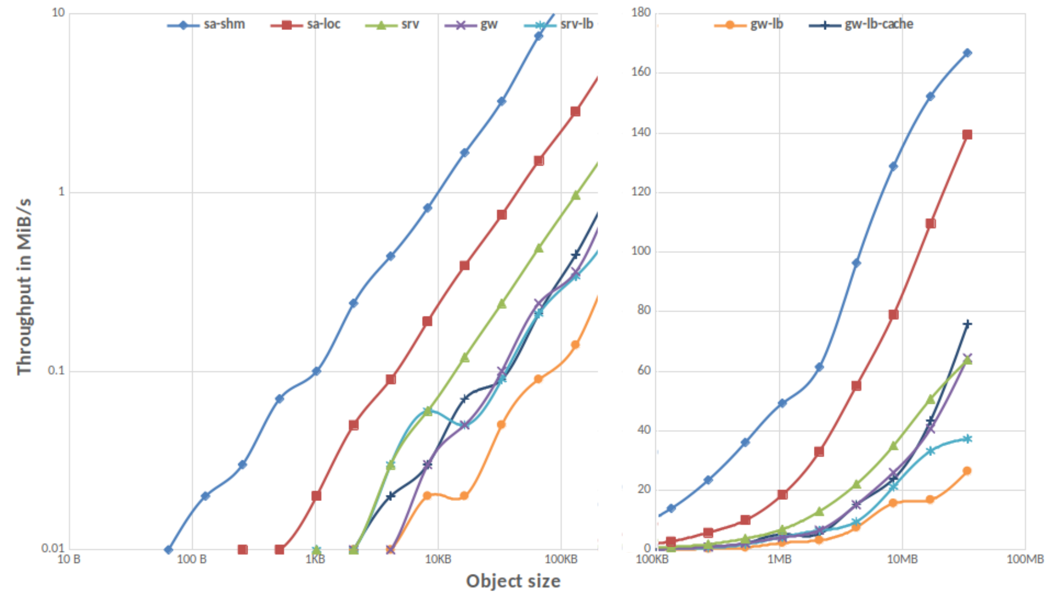


FIGURE 5.4: Write throughput for MinIO modes for 1 node and 1 PPN

Parallel I/O

We investigate to what extent parallel I/O can exploit the available network bandwidth by varying the number of clients accessing the object-store. Assuming an ideal scenario, we start MinIO in standalone mode with RAM as backend (sa-shm) on one node, while launching IOR on another set of four nodes.

Figure 5.5 and fig. 5.7 show the aggregated throughput across another four client nodes, demonstrating the scale-out throughput achieved across many tasks running on various nodes.

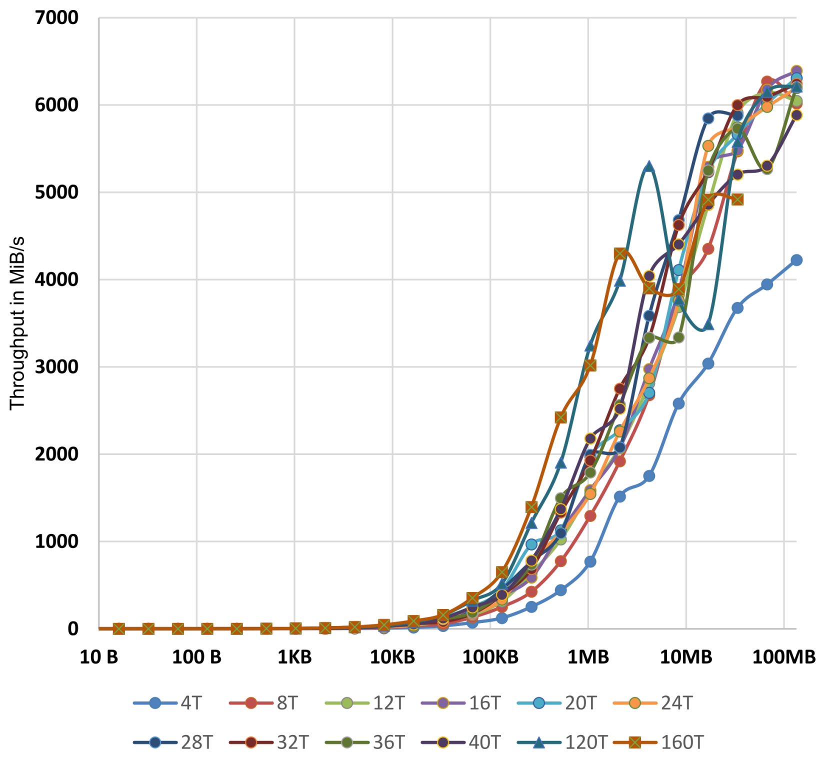


FIGURE 5.5: Read throughput for N tasks on 4 nodes

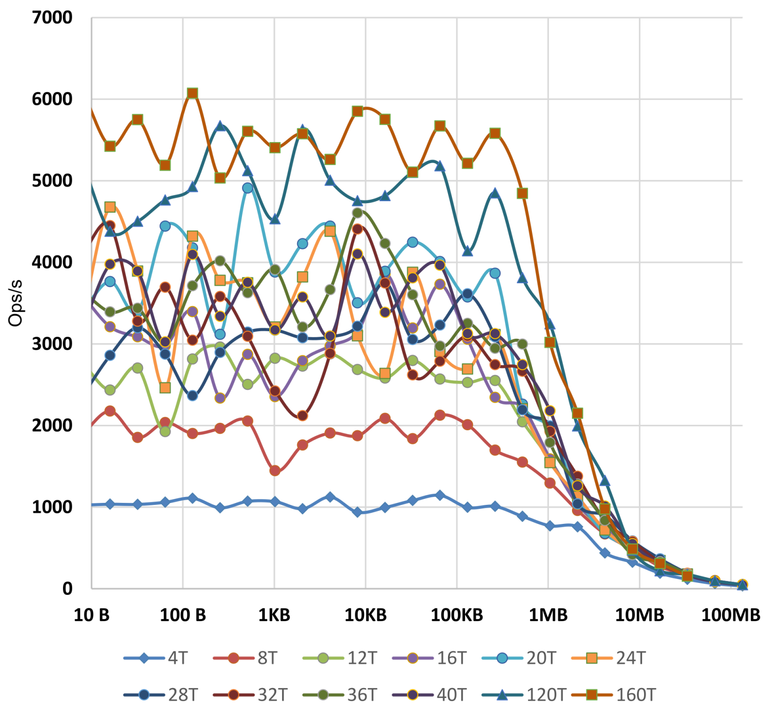


FIGURE 5.6: Read Operations/s for N tasks on 4 nodes

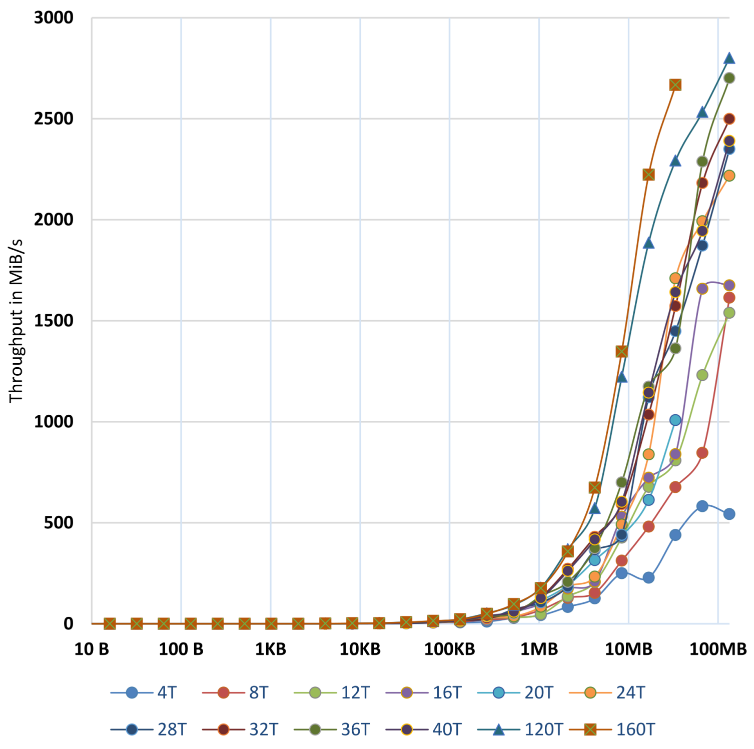


FIGURE 5.7: Write throughput for N tasks on 4 nodes

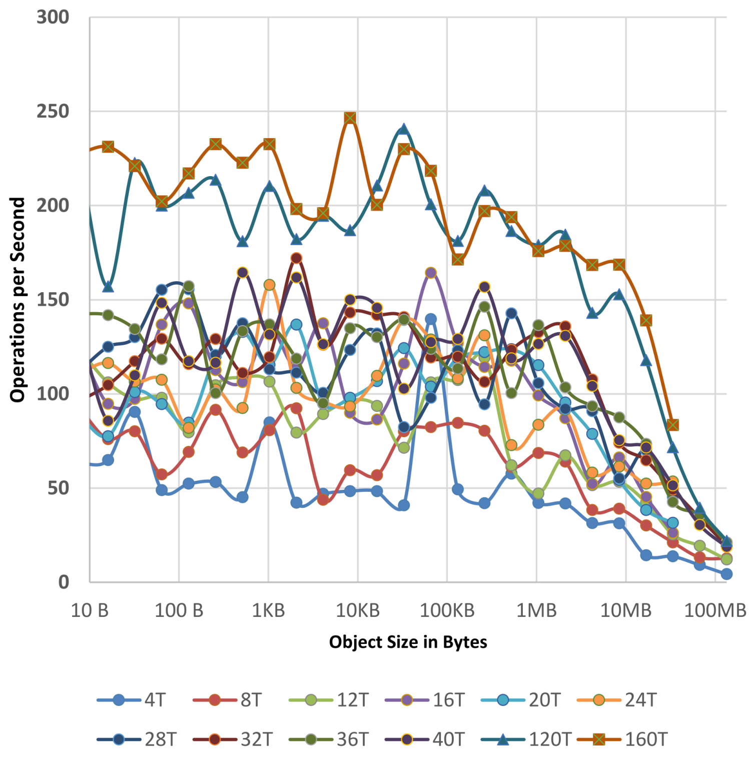


FIGURE 5.8: Write Operations/s for N tasks on 4 nodes

We notice that we achieve the best throughput when increasing the number of tasks per node (about 6000 MiB/s). The performance per client node is 1.5 GiB/s. For the single server, it is close to the available network bandwidth.

The I/O path is limited by latency. While with four threads (PPN=1), about 1000 Ops/s are achieved, with 160 threads, about 6000 Ops/s can be reached. Write achieves about 2500 MiB/s (40% efficiency) and 250 Ops/s, indicating some limitations in the overall I/O path. This also fosters splitting data larger than 1 MiB into multiple objects and using multipart (AWS, n.d.[b]) upload/download.

MinIO overhead in Gateway Mode

A more realistic scenario inside HPC is MinIO running in Gateway mode in front of Lustre.

First, we launch the benchmark on four client nodes, and we start MinIO in gateway mode on another set of nodes. We call this setup the disjoint mode.

However, this setup does not scale out efficiently. This leads us to introduce another concept, which we dub the local gateway (local-gw) mode, where MinIO is started on the N client nodes in gateway mode and uses the Lustre file system as the backend file system.

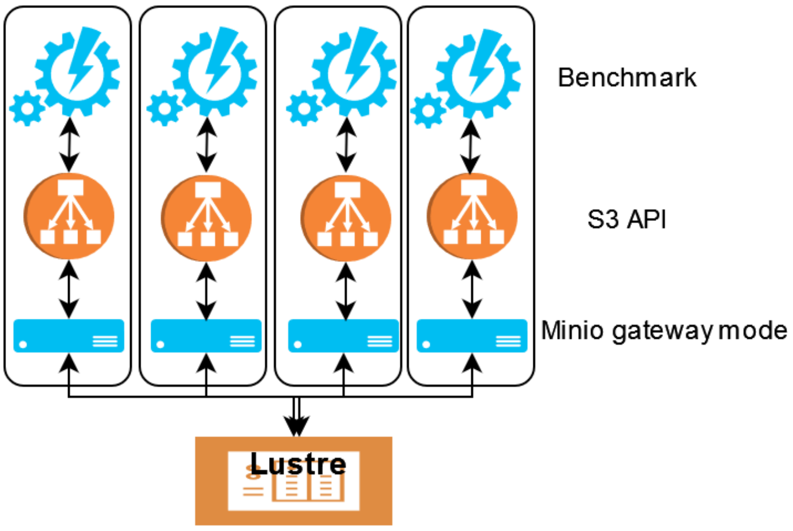


FIGURE 5.9: Benchmark against the local gateway

As depicted by fig. 5.9, we launch the benchmarks against the localhost on each node and notice that when increasing the number of tasks per node, we are achieving relatively better performance, compared to the disjoint mode, as shown in Table 5.1. However, we only achieve around 2% of Lustre's performance for various benchmarks.

Benchmark	Metric	Unit	Lustre	MinIO disjoint-gw	MinIO local-gw	local-gw % of Lustre
md-workbench	rate	IOPS	18337	37	425	2.3%
	throughput	MiBps	34.100	0.100	0.800	2.3%
IO500	ior-easy-write	GiB/s	18.671	0.153	0.286	1.5%
	mdtest-easy-write	kIOPS	5.892	0.088	0.132	2.2%
	ior-hard-write	GiB/s	0.014	0.003	0.006	45.7%
	mdtest-hard-write	kIOPS	5.071	0.036	0.076	1.5%
	ior-easy-read	GiB/s	11.475	0.693	2.071	18.1%
	mdtest-easy-stat	kIOPS	24.954	1.198	4.092	16.4%
	ior-hard-read	GiB/s	0.452	0.029	0.094	20.7%
	mdtest-hard-stat	kIOPS	18.296	1.281	3.968	21.7%
	mdtest-easy-delete	kIOPS	9.316	0.025	0.023	0.3%
	mdtest-hard-read	kIOPS	6.950	0.449	1.636	23.5%
	mdtest-hard-delete	kIOPS	4.863	0.029	0.025	0.5%

TABLE 5.1: Performance of MinIO Gateway on 4 nodes with 20 PPN

Nevertheless, we notice that increasing the number of clients and the tasks per client leads to an increase in the number of "Operation timed out" errors, which denotes a scalability issue. An issue that we address in Chapter 6.

MinIO vs REST vs TCP/IP

In chapter 4, the base performance of REST/HTTP is analyzed by emulating a best-case client/server scenario (HTTP GET Operation/Read Only Scenario from a Storage Server).

A similar setup is used here, illustrated by fig. 5.10, with nginx *version 1.18.0* as a web server and wrk running 4 threads as the benchmark tool.

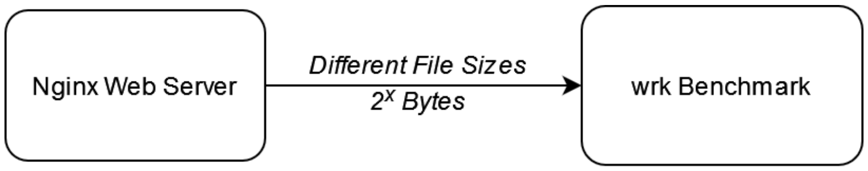


FIGURE 5.10: REST Benchmark using the Nginx web server

We conduct the same experiment on Mistral, using 1Node-4PPN and tmpfs as backend. Since the tests are accomplished against localhost, we compare the obtained results with the MinIO results running in standalone mode serving from RAM (sa-shm).

As a reference, we also include the TCP/IP performance results obtained using `iperf3`, where the server is started with four parallel client streams on the same node. The iperf3 server uses the same file sizes above the source rather than just generating random data. This feature is used for finding whether or not the storage subsystem is the bottleneck for file transfers. The number of bytes received by the client is kept equal to the file size and fits in the read buffer to best simulate the above scenario. As noted by iperf3's authors, this tool is single-threaded, and it should not be regarded as a file transfer tool; hence the values are provided for reference purpose only.

Figure 5.11 shows that for objects with size greater than 1 KB, MinIO performance – even in this ideal setup – is significantly lower than the others and that there is still room for improvement, which we address in chapter 6.

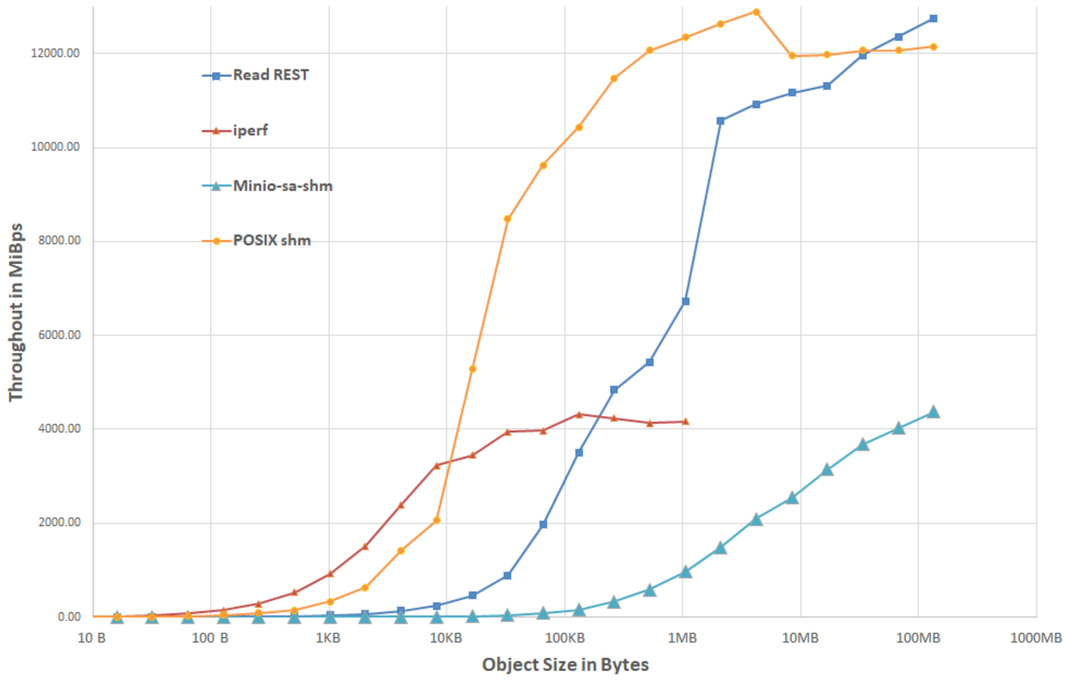


FIGURE 5.11: Read throughput Minio vs REST using 1N-4PPN

5.3.3 Test against S3 compatible systems

The next benchmarks are accomplished against different S3 compatible systems using both IO500 and MD-Workbench.

In-house Tests

The following tests are conducted against the OpenStack Swift (OpenStack-Foundation, n.d.) system already available in DKRZ, Swift version 2.19.2 is used, and the S3 interface is implemented using Swift3 version 1.12.1, which is now merged into swift middleware as the s3api; this is why only AWS signature v2 is available and as such the tests were only conducted using MD-Workbench.

On four client-nodes and with 20 PPN, a rate of 269.5 IOPs and a 0.5 MiB/s throughput are observed during the benchmark phase von MD-Workbench.

Table 5.1 also shows the IO500 results for the different systems tested inside DKRZ.

Comparison with Scality Ring

Next, we compare the performance of the MinIO obtained results in Section 5.3.2 to the results of another S3 compatible storage called Scality Ring published in (Walsdorf, n.d.).

Scality Ring is a cloud-scale, distributed software storage solution that offers a comprehensive AWS S3 REST API implementation. We are aware that this is not a fair comparison, but it gives us a qualitative comparison that validates that our results are reasonable.

In the setup described in (Walsdorf, n.d.), RING is deployed on Cisco Networking equipment much similar to the HPC network environment provided by Mistral. For the sake of simplicity, the server CPU capabilities are considered equivalent (Intel Xeon Silver 4110 vs. Intel Xeon E5-2680 v3). The published cosbench (Zheng et al., 2012) results in (Walsdorf, n.d.), from the benchmarks launched on 3 nodes with 300 threads, are compared to the MinIO Parallel I/O results described in Section 5.3.2, which are also started on 3 nodes with a total of 144 threads (no Hyperthreading). The comparison is shown in Figure 5.12.

We can see that although the write throughput is relatively similar for files having a size below 32 MB, the read throughput of MinIO is better. Scality may have an advantage for writes below 16 MiB because nearly 2x the number of threads is used. Based on these measurements, we presume that Scality does not yield a significant performance benefit over MinIO.

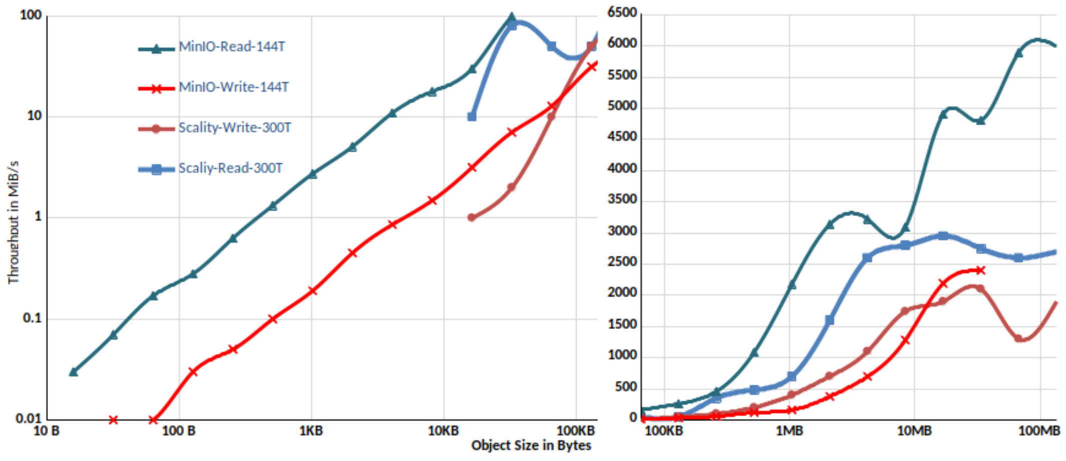


FIGURE 5.12: Throughput of Scality Ring and MinIO on 3 nodes.

Test against Cloud Systems

Different S3 vendors were contacted, which either offered a testing account or explicitly allowed us to execute the IO500 benchmark against their endpoints. We follow the guidelines depicted by the performance design patterns for Amazon S3 (AWS, n.d.[c]), especially regarding the request parallelization and horizontal scaling to help distribute the load over multiple network paths.

Since all the providers offer multi-region storage, we choose the closest storage location to Mistral (located in the EU) to ensure the lowest latency. Due to the limited number of nodes with Internet connectivity on Mistral, the benchmarks are launched on only two nodes with PPN=1. The results are summarized in Table 5.2; we scaled down the units by 1000 to better visualize the differences. The results of MinIO launched in local-gw mode – with the same number of nodes and tasks per node – are displayed in the last column.

Benchmark/System	Unit	Wasabi	IBM	Google	MinIO-local-gw
Score Bandwidth	MiB/s	0,007	1,642	0,46	12,62
ior-easy-write	MiB/s	2.35	35.00	13.35	46.39
mdtest-easy-write	IOPS	13.04	81.72	21.79	27.96
ior-rnd-write	MiB/s	0.01	0.23	0.07	1.231
mdworkbench-bench	IOPS	5.75	47.23	12.83	15.25
ior-easy-read	MiB/s	1.20	45.37	7.81	73.86
mdtest-easy-stat	IOPS	20.92	145.09	51.10	260.97
ior-hard-read	MiB/s	0.05	5.59	1.38	6.01
mdtest-hard-stat	IOPS	20.74	149.64	49.48	297.62
mdtest-easy-delete	IOPS	10.35	35.02	9.37	81.06
mdtest-hard-read	IOPS	8.54	70.06	18.90	130.36
mdtest-hard-delete	IOPS	10.28	35.25	9.48	94.32

TABLE 5.2: IO500 results comparing S3 cloud providers

The IBM cloud storage provided the best performance in our tests; however, this is far from the performance expected in HPC. Although MinIO in gateway mode provides better performance, this is also below our HPC experience since the network latency, in this case, is minimal compared to the other scenarios. A better solution is needed to leverage existing file systems found either inside or outside the HPC environment, as to be seen in chapter 6. Note that some of the mentioned providers might provide better performance when using their native interface instead of S3; however, this is outside the scope of this work. Also, the network interconnection between DKRZ and the cloud provider bears additional challenges.

Test against Huawei OceanStor Pacific 9950

Several tests were also conducted against the OceanStor Pacific 9950[6] storage system, a 5U chassis that supports up to 8 storage nodes and 80 NVMe SSDs, which provides different storage access methods, including S3 compatible access.
Within a workshop done in collaboration between Huawei and GWDG[7], the chance was given to conduct some tests against the OceanStor Pacific 9950 system within a test environment provided by Huawei and illustrated in fig. 5.13.

[6]`https://e.huawei.com/en/products/storage/distributed-storage/oceanstor-pacific-series/oceanstor-pacific-9950`

[7]Gesellschaft für Wissenschaftliche Datenverarbeitung mbH Göttingen

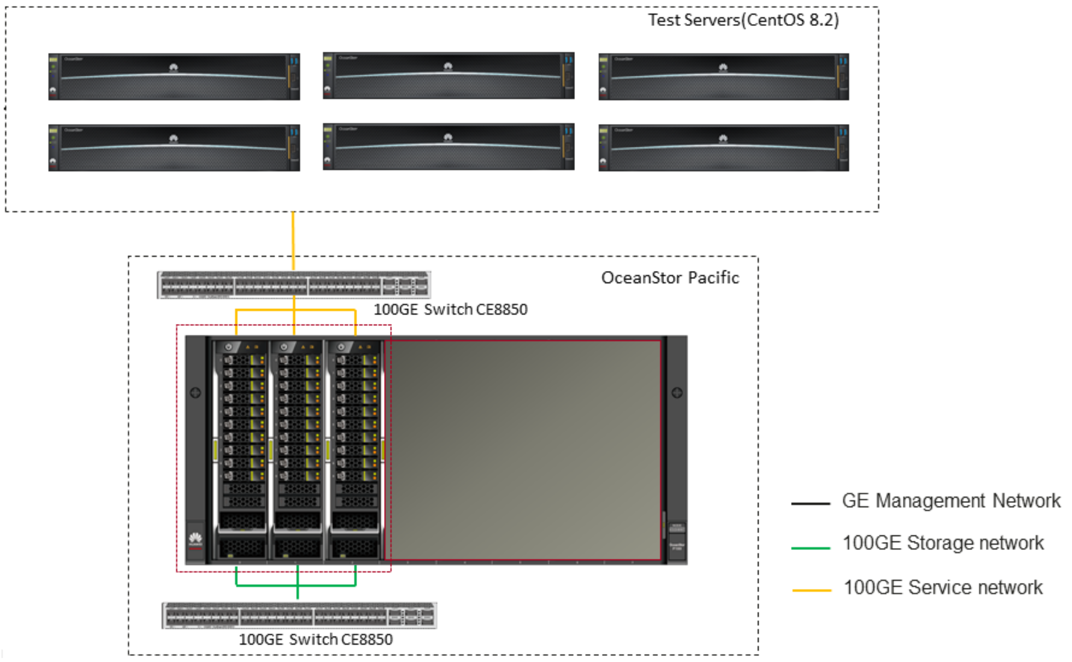


FIGURE 5.13: Test environment provided by Huawei

A round-robin load balancing between three S3 endpoints is achieved at the name resolution level of the S3 endpoint, which is done by setting a small TTL[8] of five seconds for the DNS record.

Tests were conducted against one S3 endpoint — by directly providing one IP address — and then against the endpoint name.

Since the Huawei OceanStor also provides a parallel filesystem that can be accessed using a DPC[9] mount, this was also tested using IO500. The results are depicted by table 5.3.

Benchmark	**Unit**	**S3-Single Endpoint**	**S3-Round Robin DNS**	**DPC**
ior-easy-write	GiB/s	0.70695	1.44374	4.118378
mdtest-easy-write	kIOPS	10.088596	23.343315	13.106489
ior-hard-write	GiB/s	0.161295	0.259109	-
mdtest-hard-write	kIOPS	5.574865	9.820914	1.281831
ior-easy-read	GiB/s	0.850466	1.708233	4.573681
mdtest-easy-stat	kIOPS	20.297058	36.543428	15.789467
ior-hard-read	GiB/s	0.156017	0.253328	-
mdtest-hard-stat	kIOPS	19.950588	34.737915	13.345194
mdtest-easy-delete	kIOPS	1.691461	1.65386	35.523597
mdtest-hard-read	kIOPS	8.776169	12.667338	9.576383
mdtest-hard-delete	kIOPS	1.674306	1.685916	1.114147

TABLE 5.3: IO500 results against Huawei OceanStor 9950 using 2N-40PPN

As seen, distributing the load between several S3 endpoints improves the system's overall performance.

[8]Time to Live
[9]Distributed Parallel Client

More tests are planned to be conducted against this storage system and will be posted to the git repo `https://github.com/frankgad/s3-performance-analysis-paper`, where the full results of all experiments conducted in this chapter are already published.

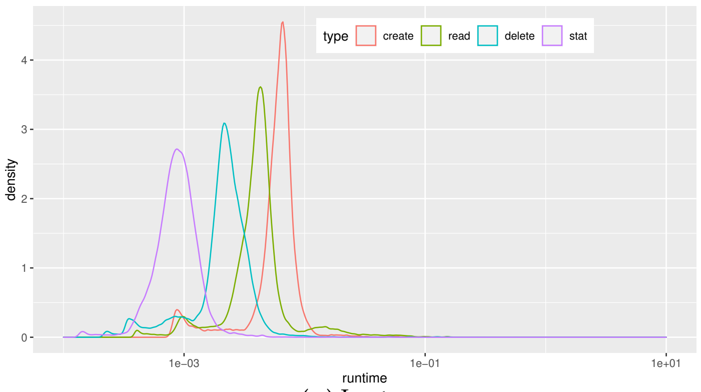
5.3.4 Latency Analysis

MD-Workbench does not only report the throughput but also the latency statistics for each I/O operation. The density of the individually timed operations is plotted as shown in fig. 5.14. A density graph can be considered a smoothed histogram where the x-axis shows the observed runtime and the y-axis represents the number of occurrences.

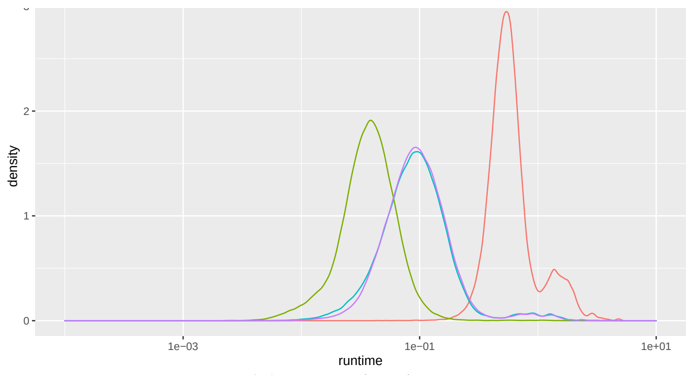
The Lustre density figure shows roughly a Gaussian distribution for individual operations, where create is the slowest operation.

Using MinIO, the creation phase takes substantially longer. The local-gw mode, as already seen from the IO500 results, yields the best performance among the tested S3 implementations; however, far from the Lustre performance, which has a latency of around 10 ms and is extremely slow compared to the network base latency of approximately 10 μs. The SwiftS3 system changes the overall behavior significantly, leading to less predictable times.

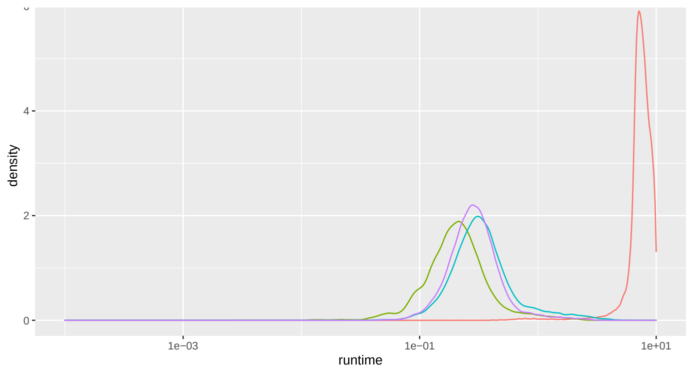
We conclude that the involved processes behind the S3 implementation are the leading cause of latency.



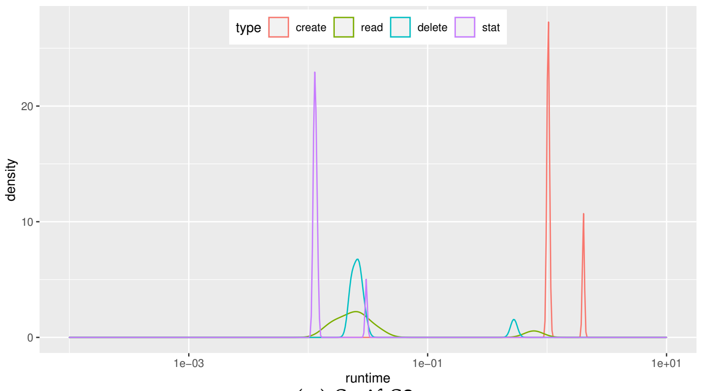
(A) Lustre



(B) MinIO local-gw



(C) MinIO disjoint



(D) SwiftS3

FIGURE 5.14: Latency density for the different systems

5.4 Summary

The S3 API is the de-facto standard for accessing cloud storage; this is why it is the component of choice when building cloud-agnostic applications. By amending IO500 to benchmark the S3 interface, we broaden the scope of the IO500 usage and enable the community to track the performance growth of S3 over the years and analyze changes in the S3 storage landscape, which will encourage the sharing of best practices for performance optimization. Unfortunately, the results indicate that S3 implementations such as MinIO are not yet ready to serve HPC workloads directly because of the drastic performance loss and the lack of scalability.

We found that the remote access to S3 is mainly responsible for the performance loss and should be addressed. Using a load balancer in front of the S3 server nodes not only represents a single point of failure but is also a bottleneck for the data transfer. We used the already existing parallel shared filesystem to leverage an S3 capable application by introducing the local gateway mode. With the advent of filesystem capable of using the node-local storage (ramdisk, SSD, NVRAM) such as GekkofsVef et al., 2020, the local-gw mode can also be extended to accommodate such underlying fs.

We conclude that S3 with any gateway mode is not yet suitable for HPC deployment as additional data transfer without RDMA support is pricey.

The next chapter will further disassemble the S3 stack to identify and overcome performance issues; thus, it introduces S3embeded, an embedded library capable of converting existing S3 application storage calls to its HPC storage equivalents.

Chapter 6

S3Embedded

In the previous chapter, we were able to pinpoint the main bottlenecks of S3. This chapter and specifically section 6.1 outlines our approach to identify the cause of the performance loss by systematically replacing parts of the popular S3 client library with lightweight replacements of lower stack components. This leads to introducing a new S3 access library, S3embedded, which proves to be highly scalable and capable of leveraging the shared cluster file systems of HPC infrastructure to accommodate several S3 client applications. Section 6.4 depicts the tests that were accomplished on Mistral and the results obtained in comparison to previous S3 implementations and Lustre. Further improvements to the S3Embedded library stack are outlined in section 6.5, taking into advantage the new evolution of the HTTP protocol or the use of MPI over RDMA[1].Section 6.7 summarises our findings and illustrates possible convergence scenarios using S3Embedded.

6.1 Performance Insights

In chapter 5, major performance issues were encountered, which can be interpreted as follows:

- A passive load balancer, i.e., without read-caches, does not offer any performance advantage; moreover, it represents a bottleneck problem reducing the scalability of the entire S3 storage system.
- The local-gw MinIO approach, depicted by section 5.3.2, produces the best results but requires the existence of a shared and commonly accessible filesystem, whereas the distributed server mode delivered the worst performance, mainly to the overhead needed to synchronize and ensure the protection of the data distributed on the different local node disks.
- The create I/O operation seems to be the most costly compared to the other I/O operations, primarily when the transfer is conducted over HTTP, i.e., disjoint mode.

The S3 stack consists of different components; fig. 6.1 shows the typical setup used by AWS.

[1]Remote Direct Memory Access

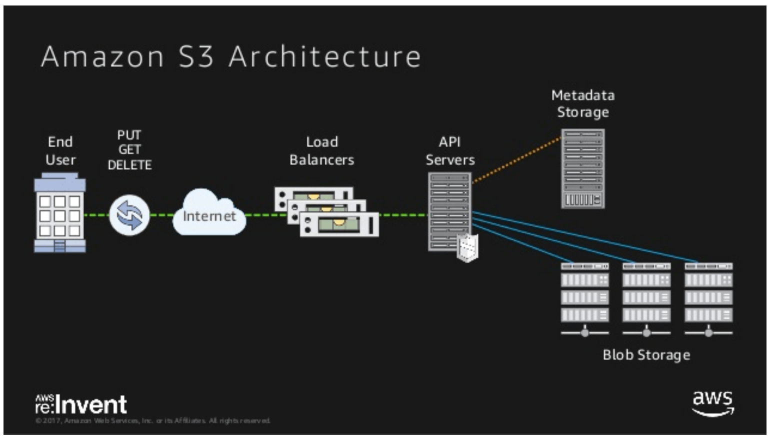


FIGURE 6.1: AWS S3 architecture presented during an AWS event

We try to disassemble it by testing different combinations where stack components are replaced or removed to optimize the performance.

- **Removing the authentication part** :

  The signature version 4 [2] used to sign the request and construct the authorization header containing the signature for the request is a process represented by four distinct steps exhibited by fig. 6.2

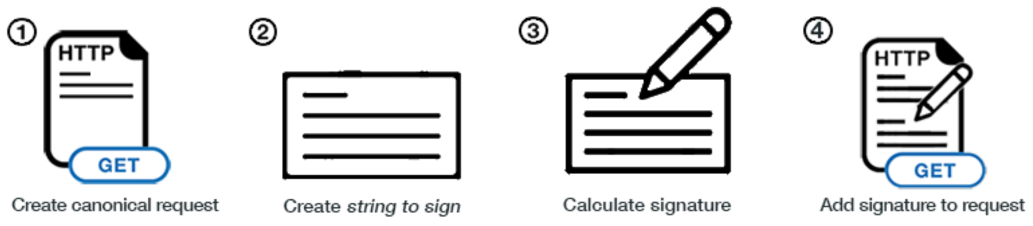


FIGURE 6.2: Request Signing using the Signature V4

  However, inside an HPC environment, the authorization and authentication are supposedly accomplished at another layer; thus, we can delegate this requirement to other systems and safely remove this mechanism.

- **Removing the REST/HTTP Transfer**: Considering the case of a shared filesystem, directly accessible from each node, we can eliminate the transfer part over REST by directly translating the S3 Calls to another I/O middleware library like POSIX or MPI-IO or even to a high-level API like the native DAOS API [3]

- **Substituting the REST/HTTP** transfer with a binary transfer by translating the I/O S3 calls as described above, the transfer, in this case, can take place over TCP, UDP, or even RDMA.

- **Substituting HTTP1.1** with newer versions of the HTTP Protocol

A flexible S3 library is needed, offering the possibility to adjust, replace or remove the components mentioned above from the software stack. For this purpose,

[2] https://docs.aws.amazon.com/general/latest/gr/sigv4_signing.html
[3] https://docs.daos.io/latest/overview/architecture

the S3Embedded[4] library is conceptualized and implemented in the following section.

6.2 S3Embedded

Since the modified benchmarks from chapter 4 use the LibS3[5] C library, a new I/O dynamic shared library called S3Embedded is introduced as a drop-in replacement of the LibS3 library; it can be used without any modification of the existing application.

Assuming the availability of a globally accessible shared file system, S3Embedded provides the following libraries:

- **libS3e.so**: This is an embedded library wrapper that converts LibS3 calls to POSIX calls inside the application address space. Any call made to the S3 interface will be directly issued to the directly attached shared storage, eliminating the need to establish a connection to a remote server; thus, the overhead due to the use of the networking protocol TCP or the transfer protocol HTTP is eliminated. The authentication and authorization are delegated to the shared filesystem and are not implemented.

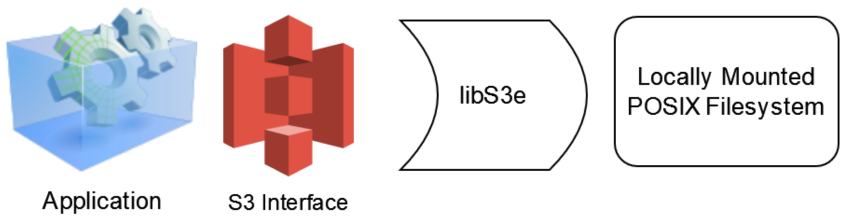


FIGURE 6.3: LibS3e Overview

At the time of writing, all the major S3 calls found in LibS3 are implemented in S3Embedded. Table 6.1 represents a mapping from a sample of some S3 calls to the corresponding Posix calls.

S3 Call	Posix Call	Description
S3_create_bucket	mkdir(path, S_IRWXU | S_IRWXG)	make directory
S3_get_object	fread(buffer, 1, byteCount, fd)	read data from the file stream into buffer
S3_delete_bucket	rmdir(path)	remove directory
S3_put_object	fwrite(buffer, 1, size, fd);	write chunks of generic data to file stream

TABLE 6.1: Mapping from a sample S3 calls to Posix calls

Some calls/functions are still not implemented because they are either delegated to the shared storage system, like those related to the security (pertaining to access control lists, ACL[6], for example, S3_get_acl[7]) or irrelevant to our purpose like those related to the data lifecycle management, for instance, S3_get_lifecycle[8].

[4]https://github.com/juliankunkel/S3embeddedlib
[5]https://github.com/bji/libs3
[6]Access Control List
[7]Gets the Acess Control List for the given bucket or object
[8]Gets the lifecycle for the given bucket

- **libS3r.so**: This library converts the LibS3 calls via a binary conversion to TCP calls which are sent through the network loopback interface to be received and handled by a local libS3-gw server application that then executes these POSIX calls, bypassing the HTTP protocol. The loopback interface is chosen to avoid any network overhead or discrepancy introduced by network components like switches and routers and to ease the comparison to the MinIO local gateway mode results obtained earlier. The libs3gw server has been designed with multi-threading for parallel processing, where a thread function named handle_connection is called to handle each connection. Whenever a request comes to the server, its main thread will create a new thread and pass the client request to that thread with its ID. Another way to implement the parallel processing is to use a non-blocking I/O approach to eliminate the overhead incurred by context switching; however, this was not implemented at the time of writing. Similar to LibS3e, authentication and authorization are disregarded since they are delegated to the shared filesystem.

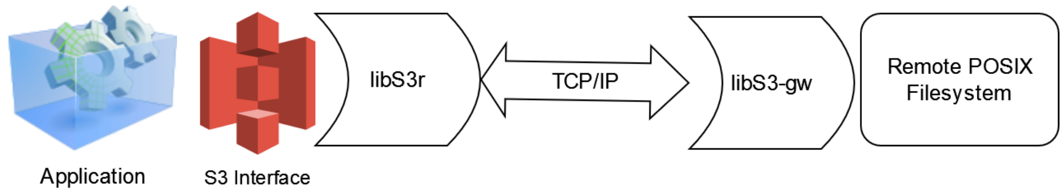


FIGURE 6.4: LibS3r Overview

By easily linking the S3Embedded library at compile time or at runtime to a libS3 compatible client application, it is possible to use the full capabilities of this library.

After compiling the library, the simplest way to use it is to add the path where it resides to the UNIX/Linux System environment variable LD_LIBRARY_PATH, therefore to tell the dynamic link loader[9] where to look for the replacement of LibS3, i.e., S3Embedded, with which the application was linked.

A typical workflow to switch between LibS3 and S3Embedded is depicted by listing 6.1.

LISTING 6.1: Commands used to use S3Embedded and switch between the multiple combinations

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/foo/ur-git/S3EmbeddedLib
$ ln -sf $PWD/libS3e.so libs3.so.4  # use S3Embedded local
$ .$PWD/libS3gw /dev/shm/ &  # start the gateway with shm as attached
    storage
$ ln -sf $PWD/libS3r.so libs3.so.4  # use S3Embedded gateway
$ ln -sf $LIBS3_PATH/libs3.so.4 libs3.so.4  # use the libs3 library
```

In practice, this can be achieved in an HPC environment by providing both LibS3 or the different implementations of S3Embedded as software modules that the data center end-users can easily use to switch between the libraries.

Due to time constraints, some functions are not fully implemented at the time of writing as in the original LibS3, like S3_initiate_multipart[10] needed to initiate

[9]small program that initiates all the applications

[10]This operation initiates a multipart upload and returns an upload ID. This upload ID is used to associate all the parts in the specific multipart upload. You specify this upload ID in each of your subsequent upload part requests

a MPU[11], and S3_copy_object_range [12], which should bring a good performance boost. Nevertheless, S3Embedded is fully S3 compatible in the sense of a drop-in library. In the following section, we use commonly available S3 compatibility tools to prove its S3 compatibility and ease of use.

6.3 S3 Compatibility Tests

Several tools are available to test the correctness and compatibility of an S3 API; two of the most commonly used tools are:

- The **s3-tests**[13] from the providers of Ceph, a set of unofficial Amazon AWS S3 compatibility tests, that might be helpful to people implementing software exposing an S3-like API. The tests utilize the Boto2 and Boto3 Python libraries. To test the S3Embedded library, which is written in the C language, we need to use the ctypes module from the Python standard library; ctypes is a foreign function library for Python that provides C-compatible data types and allows calling functions in DLLs or shared libraries by wrapping these libraries in pure Python. However, this can be time-consuming since special attention should be given to provide the expected typed parameters when calling a specific C function; this is without considering the performance limitations [14].

- **Mint**[15] from MinIO is a testing framework that was initially written to test the functionality of the Minio object server. It runs correctness, benchmarking, and stress tests using several SDK[16]s and tools. It is mainly written in Go; however, it also provides a set of tests that use the awscli[17], specifically the *aws s3* command, which is the command-line interface provided by AWS to interact with the AWS S3 storage. The tests were adjusted to use the "s3" command, which is the command-line executable provided by the LibS3, and by using the linking method described above, the "s3" tool can be linked to the S3Embedded library, in order to be used in the S3verify script.

LISTING 6.2: S3 Compatibilty Tests against a MinIO public test server and against S3Embedded

```
# Against MinIO Test Endpoint using the original LibS3
$ export S3_HOSTNAME="play.minio.io:9000"
$ ./s3verify.sh
$ cat outs3.log
{"name": "s3", "duration": "1612", "function": "test_make_bucket", "
    status": "PASS"}
{"name": "s3", "duration": "189", "function": "test_make_bucket_error
    ", "status": "PASS"}
{"name": "s3", "duration": "22012", "function": "test_put_object", "
    status": "PASS"}
{"name": "s3", "duration": "394", "function": "test_put_object_error"
    , "status": "PASS"}
```

[11] S3 MultiPart Upload

[12] Copies portion of an object from one location to another. The object may be copied back to itself, which helps replace metadata without changing the object. Required when doing greater than 5GB object copies

[13] https://github.com/ceph/s3-tests

[14] http://tungwaiyip.info/blog/2009/07/16/ctype_performance_benchmark

[15] https://github.com/minio/mint

[16] Software Development Kit

[17] https://aws.amazon.com/cli/

```
{"name": "s3", "duration": "185532", "function": "
    test_put_object_multipart", "status": "PASS"}
{"name": "s3", "duration": "26369", "function": "test_get_object", "
    status": "PASS"}
{"name": "s3", "duration": "327153", "function": "
    test_get_object_multipart", "status": "PASS"}
{"name": "s3", "duration": "226442", "function": "
    test_sync_list_objects", "status": "PASS"}
# Against S3Embedded with tmpfs as Storage Backend
$ ln -sf $PWD/libS3e.so libs3.so.4
$ export S3_HOSTNAME="/dev/shm/"
$ ./s3verify.sh
$ cat outs3.log
{"name": "s3", "duration": "145", "function": "test_make_bucket", "
    status": "PASS"}
{"name": "s3", "duration": "40", "function": "test_make_bucket_error"
    , "status": "PASS"}
{"name": "s3", "duration": "83", "function": "test_put_object", "
    status": "PASS"}
{"name": "s3", "duration": "46", "function": "test_put_object_error",
     "status": "PASS"}
{"name": "s3", "duration": "551", "function": "
    test_put_object_multipart", "status": "PASS"}
{"name": "s3", "duration": "309", "function": "test_get_object", "
    status": "PASS"}
{"name": "s3", "duration": "753", "function": "
    test_get_object_multipart", "status": "PASS"}
{"name": "s3", "duration": "1466", "function": "
    test_sync_list_objects", "status": "PASS"}
```

The tests are split into two parts:

– The first part of listing 6.2 illustrates the compatibility tests against an S3 test endpoint, provided by the MinIO project, and using the LibS3 library, i.e., using the "s3" executable found within the LibS3 library.

– The second part shows that, after switching the "s3" executable to use S3Embedded, the same tests are accomplished successfully against the local tmpfs filesystem */dev/shm*. As expected, we can see that the duration of each test is obviously smaller.

6.4 S3Embedded HPC Tests

6.4.1 S3embedded vs MinIO vs REST vs TCP/IP

Building on the results obtained in section 5.3.2, we conduct the same experience using S3Embedded for both local and remote implementation.

The tests are also conducted on Mistral, using 1Node-4PPN and tmpfs as backend. The TCP/IP performance results obtained using `iperf3` are also included for reference purposes.

Figure 6.5 shows that the performance of S3Embedded is the best among the S3 implementation, and it is the closest to the direct filesystem access, represented by "POSIX shm", however, the remote implementation or LibS3r, which is presumed to be comparable to the REST performance, seems not to perform very well.

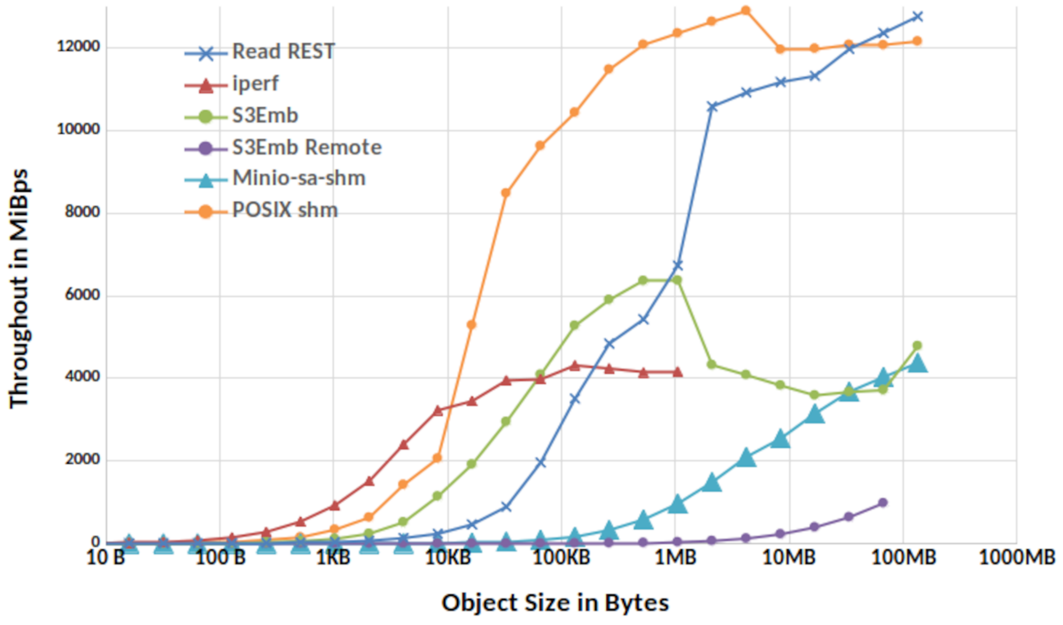


FIGURE 6.5: Read throughput S3Embedded vs Minio vs REST using 1N-4PPN

6.4.2 S3Embedded IOR Results

In Figure 6.6, we display the results of the **IOR** Benchmark while using the libraries mentioned above, in comparison with direct Lustre access and MinIO operating in the local-gw mode already described in Section 5.3. Note that some values are missing in the MinIO-local-gw results, although we repeated the benchmark several times. This is because this setup does not scale well with the number of clients, as noted in Section 5.3.

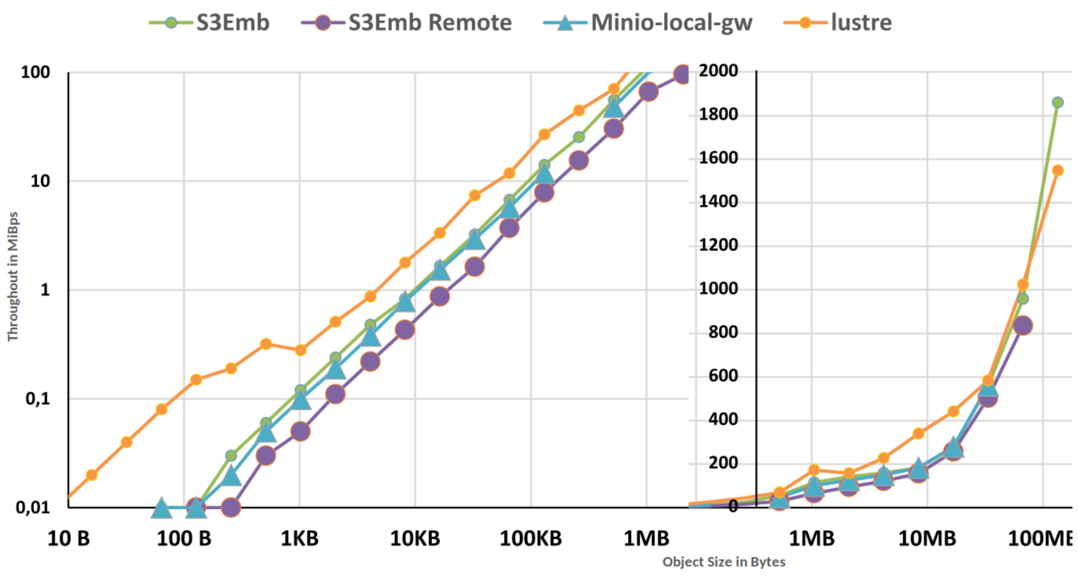


FIGURE 6.6: Read throughput of S3Embedded vs Lustre vs MinIO for 5N-20PPN

Using S3Embedded helped us to pinpoint a performance problem in the IOR S3 plugin: we noticed that the delete process in IO500 is time-consuming since when trying to delete a bucket, our developed IOR S3 plugin tries to list the content of the entire bucket – calling `S3_list_bucket()` – for each file to be deleted to clean the fragments; however, since, in case of S3Embedded, all files are actually placed in a single directory, this ought to be very time-consuming. One workaround is to use the option bucket-per-file, which effectively creates a directory per file. However, since this workaround does not cover all test workloads in the IO500, an environment variable called "S3LIB_DELETE_HEURISTICS", specific to the IOR S3-plugin, is introduced. It defines the file size threshold of the initial fragment above which the list_bucket operation is to be executed; otherwise, a simple S3_delete_object is performed. While this optimization is not suitable for a production environment, it allows us to determine the best-case performance for using S3 with the IO500 benchmark.

The results delivered by S3Embedded are very close to the ones obtained for the Lustre direct access — mainly for files larger than 32 MB — and also far superior to those supplied by MinIO local-gw; they are also free from timeout errors.

6.4.3 S3Embedded IO500 Results

The results of the **IO500** benchmark, displayed in table 6.2, reflect the performance improvement delivered by S3Embedded/S3Remote and reveal the full potential of using such wrapper libraries.

System	Unit	MinIO-local-gw	Lustre	S3Embedded	S3Remote
ior-easy-write	GiB/s	0.14	5.47	0.61	0.69
mdtest-easy-write	kIOPS	0.09	7.97	2.42	3.13
ior-easy-read	GiB/s	0.32	2.78	0.48	0.42
mdtest-easy-stat	kIOPS	0.85	13.82	8.02	6.94
ior-hard-read	GiB/s	0.019	0.139	0.046	0.042
mdtest-hard-stat	kIOPS	0.86	5.10	7.25	6.65

TABLE 6.2: IO500 results for S3Embedded and S3Remote compared to MinIO-local-gw and lustre using 2N-5PPN

Benchmark/System	Unit	MinIO-local-gw	Lustre	S3Embedded	S3Remote
ior-easy-write	GiB/s	0.75	23.49	3.40	1.99
mdtest-easy-write	kIOPS	0.39	17.11	7.52	1.19
ior-hard-write	GiB/s	0.01	0.04	0.30	0.05
mdtest-hard-write	kIOPS	0.10	7.25	3.41	0.55
ior-easy-read	GiB/s	2.46	15.87	2.40	1.36
mdtest-easy-stat	kIOPS	5.09	42.59	28.53	0.62
ior-hard-read	GiB/s	0.11	0.38	0.11	0.02
mdtest-hard-stat	kIOPS	4.37	31.49	26.66	0.60
mdtest-easy-delete	kIOPS	-	9.15	5.98	0.41
mdtest-hard-read	kIOPS	-	6.34	3.82	0.29
mdtest-hard-delete	kIOPS	-	6.27	5.04	0.41

TABLE 6.3: IO500 results for S3Embedded and S3Remote compared to MinIO-local-gw and lustre using 5N-20PPN

We notice that Lustre's POSIX performance is often more than 10x faster than the MinIO-local-gw mode, which exhibits a scalability problem since the error rate increases along with the number of Nodes/PPN. In fact, the scalability of MinIO-local-gw hits its limit with the combination of 5 Nodes and 20 PPN. Even at this point, the results contain many errors, and many iterations are conducted to just fill the values in the table.

In contrast, the S3Embedded wrappers deliver much better performance, which is closer to Lustre's native performance. They are more resilient to the number of clients, as shown in table 6.3, i.e., much more scalable than the MinIO-local-gw setup.

Benchmark/System	Unit	Lustre	S3Embedded	S3Remote
ior-easy-write	GiB/s	3.202073	0.990504	0.827451
mdtest-easy-write	kIOPS	13.548102	11.819631	0.240857
ior-hard-write	GiB/s	0.015462	0.215551	0.010041
mdtest-hard-write	kIOPS	5.615789	2.226332	0.111696
ior-easy-read	GiB/s	7.483045	0.375538	0.311502
mdtest-easy-stat	kIOPS	21.884679	20.489986	0.123976
ior-hard-read	GiB/s	0.095884	0.022462	0.005101
mdtest-hard-stat	kIOPS	17.428926	6.90457	0.125178
mdtest-easy-delete	kIOPS	8.995095	7.77739	0.10289
mdtest-hard-read	kIOPS	0.184843	1.287055	0.249273
mdtest-hard-delete	kIOPS	8.108844	6.711771	0.249282

TABLE 6.4: IO500 results for Lustre vs S3Embedded using 10N-1PPN

Even with 10 or 50 Nodes, as shown in table 6.4 and table 6.5 respectively, the S3embedded library can handle the extra load and yields a performance closer to Lustre, although a performance gap remains.

Benchmark/System	Unit	Lustre	S3Embedded	S3Remote
ior-easy-write	GiB/s	16.262279	5.363676	1.516302
mdtest-easy-write	kIOPS	18.104786	15.952838	1.111315
ior-hard-write	GiB/s	0.030967	0.375326	0.032984
mdtest-hard-write	kIOPS	13.705966	4.337179	0.361589
ior-easy-read	GiB/s	43.390676	2.817122	1.309052
mdtest-easy-stat	kIOPS	46.662299	45.878814	0.620202
ior-hard-read	GiB/s	0.218977	0.128423	0.025549
mdtest-hard-stat	kIOPS	43.834921	44.443974	0.62586
mdtest-easy-delete	kIOPS	9.322632	9.262801	0.585572
mdtest-hard-read	kIOPS	4.079555	6.673985	1.246393
mdtest-hard-delete	kIOPS	8.457431	6.377992	1.246105

TABLE 6.5: IO500 results for Lustre vs S3Embedded using 50N-1PPN

We notice that the results of some metadata tests show a slightly better performance in the case of S3Embedded than Lustre, although for both Lustre and S3Embedded, the `stat()` call is used. This might be due to the way S3Embedded implements `S3_test_bucket()`[18], where the size and rights for the directory and not

[18]Tests the existence of an S3 bucket, additionally returning the bucket's location if it exists and is accessible

of the actual file are captured, which seems to be faster.

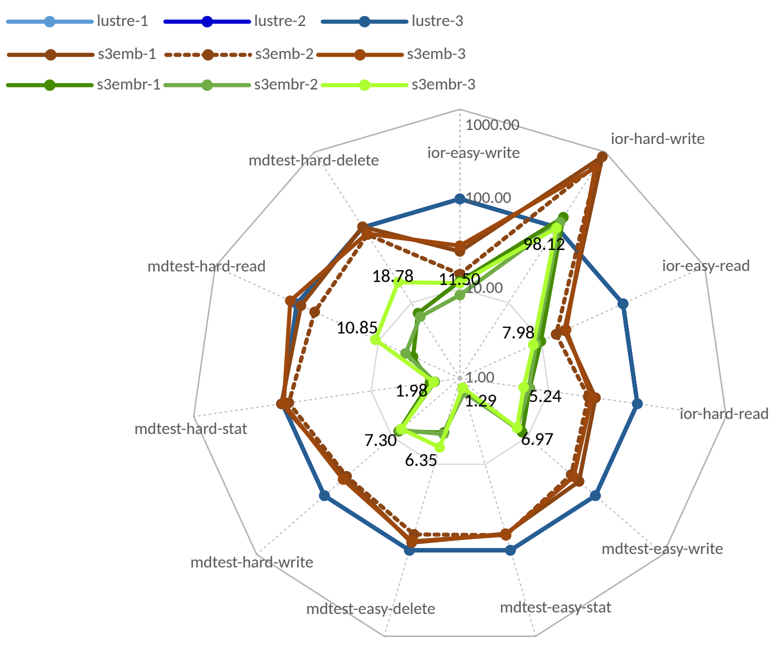


FIGURE 6.7: IO500 results of different runs using 5N-20PPN

The radar chart in fig. 6.7 shows the relative performance of S3embedded and S3remote in % for three independent runs of all benchmarks. Note that the three Lustre runs are so similar that they overlap in the figure. The graph clearly shows the performance gaps between the two implementations of the S3Embedded library. To ease and clarify the comparison, the relevant performance numbers for the run entitled *s3embr-3* are displayed. By comparing the Lustre results with the ones from LibS3r, we notice that only for the *ior-hard-write* test, the performance ratio is close to 100% while it usually achieves around 5 to 10% of Lustre performance. On the other hand, the LibS3e library delivers much better results, however, it also lacks performance for some benchmarks.

6.5 S3Embedded Possible Optimization

After establishing the **performance feasibility** of the S3Embedded library and revealing its ease of use leading to a low **administrative overhead**, we focus in the following sections on the possible optimization of this library by further substituting some components of the S3 stack with presumably more performant alternatives.

6.5.1 S3Embedded over HTTP2 or HTTP3

The new techniques introduced in the more recent versions of the HTTP protocol like the use of a small number of connections, multiplexing the HTTP datagram, compressing the header, or allowing the server to "push" data pro-actively to the client while eventually using UDP[19] to accomplish these, bear the potential to improve the performance of S3Embedded.

[19]User Datagram Protocol

Although a full implementation of the remote S3Embedded library using HTTP2 or HTTP3 is possible, it is outside the scope of this thesis. A more feasible implementation, inspired by the HTTP3 protocol, consists of replacing the TCP transfer with a connectionless alternative which is UDP, as illustrated in fig. 6.8.

Implementing the remote connection over UDP introduces many obstacles that need to be addressed.

- UDP is a connectionless and unreliable transfer protocol; the packets are sent without any guarantee of delivery and any info to help the client find out the correct order.
- The size of the packet also depends on the MTU[20] of the network, which represents the size of the largest datagram — UDP packets are also called datagrams — that can be sent over the network.

Since the communication between the S3embedded client library and the gateway should be reliable, two UDP streams are opened to allow the communication between both sides, and a first approach to handle datagram fragmentation is also implemented[21]. Other features found in TCP and implemented in HTTP3 like retransmissions of lost packets, congestion control, ordering, and connection ids were not implemented at the time of writing.

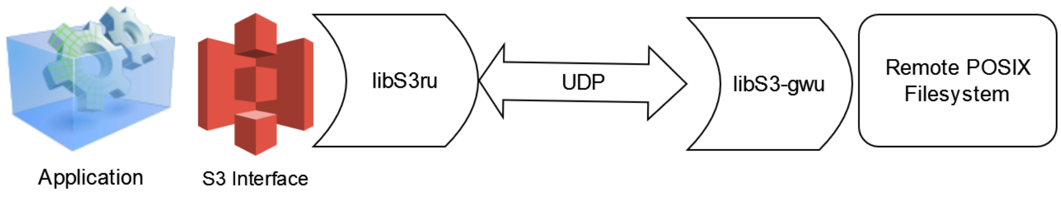


FIGURE 6.8: LibS3r over UDP

IOR is used to test the performance introduced by using UDP instead of TCP; a sample run is depicted by listing 6.3

LISTING 6.3: IOR tests againt the UDP implementation of S3Embedded in comparison with the other implementations

```
## Tests against S3embedded Remote over UDP
$ ./build/bin/ior -a=S3-libs3 --S3-libs3.access-key="1" --S3-libs3.secret-
    key="1" --S3-libs3.host=/dev/shm/test -i 3 -b 4096 -t 4096
IOR-3.4.0+dev: MPI Coordinated Test of Parallel I/O
## Output omitted
Results:

access     bw(MiB/s)  IOPS        Latency(s)  block(KiB)  xfer(KiB)   open(s)
       wr/rd(s)    close(s)    total(s)    iter
------     ---------  ----        ----------  ----------  ---------   --------
       --------    --------    --------    ----
write      2.63       1245.34     0.000803    4.00        4.00        0.000669
       0.000803    0.000004    0.001483    0
read       2.72       1297.34     0.000771    4.00        4.00        0.000657
       0.000771    0.000003    0.001438    0
write      2.35       1116.10     0.000896    4.00        4.00        0.000756
       0.000896    0.000003    0.001661    1
```

[20]Maximum Transmission Unit

[21]Please refer to the github Repo https://github.com/juliankunkel/S3embeddedlib for details regarding the source code

```
read      2.99       1650.00     0.000606    4.00        4.00        0.000686
        0.000606   0.000005   0.001305    1
write     2.24        954.99     0.001047    4.00        4.00        0.000687
        0.001047   0.000003   0.001743    2
read      3.12       1628.86     0.000614    4.00        4.00        0.000631
        0.000614   0.000003   0.001254    2

## Tests against SEmbedded Remote over TCP
$ ./build/bin/ior -a=S3-libs3 --S3-libs3.access-key="1" --S3-libs3.secret-
    key="1" --S3-libs3.host=/dev/shm/test -i 3 -b 4096 -t 4096
IOR-3.4.0+dev: MPI Coordinated Test of Parallel I/O
## Output omitted
Results:

access    bw(MiB/s)   IOPS       Latency(s)  block(KiB)  xfer(KiB)   open(s)
        wr/rd(s)    close(s)    total(s)    iter
------    ---------   ----       ----------  ----------  ---------   --------
        --------   --------   --------    ----
write     2.83       1154.82     0.000866    4.00        4.00        0.000507
        0.000866   0.000004   0.001382    0
read      4.27       2082.57     0.000480    4.00        4.00        0.000426
        0.000480   0.000003   0.000914    0
write     3.66       2109.81     0.000474    4.00        4.00        0.000587
        0.000474   0.000002   0.001068    1
read      3.57       1612.57     0.000620    4.00        4.00        0.000467
        0.000620   0.000003   0.001094    1
write     3.30       1655.86     0.000604    4.00        4.00        0.000571
        0.000604   0.000003   0.001182    2
read      3.99       2114.06     0.000473    4.00        4.00        0.000497
        0.000473   0.000003   0.000978    2

## Tests against SEmbedded
$ ./build/bin/ior -a=S3-libs3 --S3-libs3.access-key="1" --S3-libs3.secret-
    key="1" --S3-libs3.host=/dev/shm/test -i 3 -b 4096 -t 4096
IOR-3.4.0+dev: MPI Coordinated Test of Parallel I/O
## Output omitted
Results:

access    bw(MiB/s)   IOPS       Latency(s)  block(KiB)  xfer(KiB)   open(s)
        wr/rd(s)    close(s)    total(s)    iter
------    ---------   ----       ----------  ----------  ---------   --------
        --------   --------   --------    ----
write     8.84       7463        0.000134    4.00        4.00        0.000301
        0.000134   0.000002   0.000442    0
read      50.10      18157       0.000055    4.00        4.00        0.000018
        0.000055   0.000002   0.000078    0
write     21.82      10866       0.000092    4.00        4.00        0.000080
        0.000092   0.000002   0.000179    1
read      44.89      16913       0.000059    4.00        4.00        0.000021
        0.000059   0.000003   0.000087    1
write     17.36      8339        0.000120    4.00        4.00        0.000098
        0.000120   0.000003   0.000225    2
read      52.68      21183       0.000047    4.00        4.00        0.000020
        0.000047   0.000003   0.000074    2
```

Tests conducted with other file sizes also bring us to the same conclusion: The use of UDP did not convey any performance benefit in our use case.

In the case of HTTP3, the use of UDP was mainly introduced to overcome the TCP head of line blocking issue found in error-prone networks with a relatively high packet loss (Perna et al., 2022); however, implementing the reliability stack on top of it is not just a CPU and memory consuming measure, as seen in section 4.4.1, but also adds a complexity that increases the integration and administrative overhead without providing any advantage since typical HPC network are highly performant and reliable.

6.5.2 S3Embedded over RDMA

As seen in chapter 4, the transfer over TCP/IP exhibits a higher CPU overhead mainly due to the packet handling packets at the Operating system kernel level; it also shows a higher message transfer latency in comparison to the data transfer over RDMA[22].

The RDMA transfer is achieved by performing direct memory access from one node to another without involving the operating system on the destination node. Its benefits over TCP/IP can be resumed as follows:

- The data is transferred from the sending application directly to the receiving application memory area without involving the operating system of the destination node or its network interface stack.
- Avoiding memory copies on both sender and receiver, providing the application with the most negligible round trip latency and lowest CPU overhead.
- RDMA transmits data as messages, while TCP sockets transfer data as a stream of bytes. RDMA does not require any extra headers like the one employed in the TCP stream that consumes additional network bandwidth and processing.
- RDMA protocol is inherently asynchronous; i.e., no blocking is required during a message transfer.

The use of RDMA instead of TCP for the transfer part of the remote implementation of the S3Embedded library, as shown in fig. 6.9, can eventually lead to a lower CPU overhead and lower network message latency.

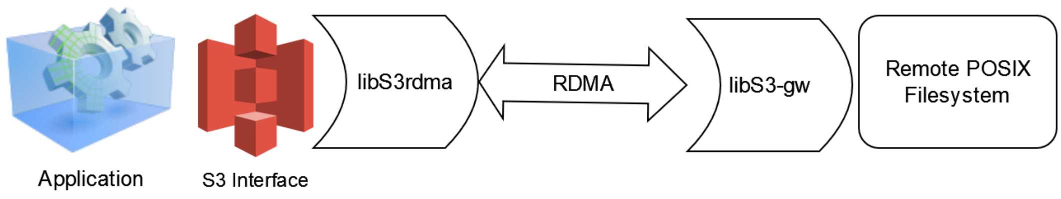


FIGURE 6.9: LibS3r over RDMA

We can use several frameworks to achieve this purpose, for example:

- libfabric[23], which is part of Open Fabrics Interfaces (OFI) and implements RDMA communications through verbs interfaces.
- The Unified Communication X (UCX)[24] framework exposes a set of abstract communication primitives utilizing the most appropriate hardware resources and offloads. RDMA (InfiniBand and RoCE), TCP, GPUs are some examples.

[22]Remote Direct Memory Access
[23]https://github.com/ofiwg/libfabric
[24]https://openucx.readthedocs.io/en/master/index.html

6.6 Convergence Scenarios using S3Embedded

Based on the convergence assessment model illustrated in chapter 3, the use of cloud storage inside an HPC environment is one of the most promising approaches to foster HPC and Cloud convergence.

Developing a storage API is nevertheless a challenging endeavor. Many criteria should be filled to ease the adoption and avoid issues during the implementation. Good documentation, solid design, security, scalability, and flexibility are just a few aspects to consider.

The S3 API has a huge user base and market share, it is well documented, and it is easy to find different implementations in various programming languages. It is also a mature API offering many optimizations, for example:

- Multipart upload allows uploading a single object as a set of parts. This leads to an increase in the transfer throughput by smartly allocating the available bandwidth.
- S3 Select allows the use of SQL[25] statements to filter the contents of an S3 object and retrieve just the subset of required data, and as such, reducing the amount of transferred data and decreasing the cost and latency to retrieve the data.

So instead of re-inventing the wheel, it is reasonable to use the S3 interface for any application intended to be used in an HPC Cloud converged system.

S3Embedded or similar wrapper libraries allow any application equipped with an S3 interface to be executed in an HPC environment without any code modification and even without the setup of a cloud storage solution on-premises. The application can seamlessly use the existing shared storage system found in the HPC environment without any performance loss. Moreover, the application will benefit from a performance gain due to the better I/O performance from S3Embedded in comparison to the cloud solution, as seen in section 6.4.3, not to mention the compute and network advantage gained from using an HPC environment.

The use of the S3 interface is standard for cloud applications and particularly for Big-Data applications running in the cloud, for example, Amazon Elastic MapReduce [26], used for running Big-Data frameworks, such as Apache Hadoop [27] and Apache Spark [28], recommends the use of S3 interface as part of the user workflow.

With a suitable S3 wrapper implementation similar to S3Embedded, it is possible to make cloud applications and cloud big-data frameworks highly portable and allow them to benefit from the converged HPC platform; these benefits can be summarised as follows:

- Applications portability and platform independence.
- Avoiding any vendor lock-in.
- HPC network optimization, by eliminating long network paths and hops.
- Minimal administrative overhead due to the dynamic linking of the suitable library, and as such providing a basis for agile services development and leveraging services provisioning.

[25]Simple Structured Query Language
[26]https://docs.aws.amazon.com/emr
[27]https://hadoop.apache.org
[28]https://spark.apache.org

- Simultaneous use of cloud and HPC storage systems, in a sense that some results can be copied into a public provider S3 compatible storage to ease the sharing of results in a secure manner.
- A wrapper library can be easily extended to deliver *performance metrics*; in the case of S3Embedded, the duration of each S3 call can be recorded and saved to a time-series database in order to gain helpful performance insights into the underlying storage system.

Another use case for an S3Embedded python implementation is to leverage the Highly Scalable Data Service (HSDS) (HDF-Group, n.d.) introduced in chapter 2, which is is the Object Store-Based Data Service for Earth System Science, providing all the functionality that the HDF5 library traditionally offers however using cloud-based storage (e.g., AWS S3).

6.7 Summary

By introducing S3Embedded, a new lightweight drop-in replacement for LibS3, we investigated the cause of the performance issues encountered in the previous chapter while providing a roadmap toward Cloud-HPC agnostic applications that can be seamlessly executed in the public cloud or HPC.

The analysis also proves that there can be a performance convergence — at the storage level — between Cloud & HPC over time by using a high-performance S3 library like S3Embedded.

The provided compatibility tests described in section 6.3 not only help us demonstrate the validity of the S3Embedded implementation but are also of great help to the community for optimizing S3Embedded and porting it to other programming languages or even creating similar wrapper libraries.

The results obtained when comparing S3embedded to Lustre and MinIO prove its **performance feasibility**. On the other hand, its ease of use leads to a low **administrative overhead**, and the possibility to use existing storage without any additional hardware investments demonstrates its **cost-efficiency**.

In fact, one of the benefits of utilizing wrapper libraries like S3Embedded is its versatility; no special hardware or software is required to use it fully. This helps avoid vendor lock-in, which is, unfortunately, one of the most common problems in HPC and Cloud landscapes. Hence, S3embedded is a vendor-agnostic abstraction layer located between the application and the available storage in the converged ecosystem, and by offering the S3 cloud API standard, it enables highly portable applications that can be seamlessly run in the cloud and in HPC, overcoming as such the problem of **Data Locality**[29] faced when the data is not portable due to its massive size, time or cost constraints.

Furthermore, standards take a long time and much effort to develop, so instead of introducing new standards, the use of already established and mature standards is the best and most cost-effective way to bring forward the scientific community and achieve the business goals of any enterprise.

As such, the use of wrapper libraries fits perfectly with the convergence assessment model defined in chapter 3.

[29]Moving computation to the node where that data resides. `https://www.thoughtworks.com/insights/decoder/d/data-locality`

Chapter 7

Conclusion and Future Work

This chapter summarizes the work conducted in this thesis. The most significant results obtained are recapped, and the novelty of this work is outlined. Future work and possible improvements are also proposed.

7.1 Achieved Status

This thesis focuses on the convergence between HPC and Cloud. After laying the ground for this work in chapter 1, chapter 2 presents the terms and technologies related to this work and provides a comparison matrix between the relevant storage technologies. One of the high-level questions raised in section 1.3.1 is also answered in this chapter:

☐ What defines a converged system?

■ The term *HPC and Cloud Convergence* seems to be subjectively interpreted: for a cloud provider or cloud service broker, it just means HPC Cloud, i.e., running HPC Workloads in the cloud, from the point of view of an HPC data center supplier or vendor, it is the other way around. This work objectively defines the term convergence: the primary purpose of convergence is the possibility for different technological systems to emerge toward performing similar tasks. The resulting converged system can seamlessly accomplish any types of tasks that were previously completed on only one of the past non-converged systems. Hence, a converged HPC and Cloud system can seamlessly run both HPC and Cloud applications.

Chapter 2 also outlines the HTTP protocol evolution and describes the various HPC and Cloud convergence scenarios, based on the definition mentioned above.

Chapter 3 explains the research methodology used in this work, and by introducing a full-featured convergence assessment model, it helps us answer other questions posed in section 1.3.1:

☐ How to assess the level of HPC and Cloud convergence?

■ Section 3.2 outlines the assessment methodology needed to measure the extent of the convergence achieved by a converged system. This model consists of different components, namely the **performance feasibility**, the **administrative overhead** and the **cost efficiency**, which are explored in depth. This model is used to compare the different solutions presented in chapter 2. Table 3.4 illustrates the results for each scenario side-by-side and shows that using object storage inside HPC is one of the most promising approaches to leverage the HPC Cloud convergence.

☐ Can we use HPC and cloud storage technologies concurrently? Which workflows will benefit from such settings, which I/O interfaces are suitable?

■ The response to these questions spans over different chapters, from table 3.4 we find that when using **Cloud Storage in HPC**, the **administrative overhead** is kept moderate once the application's compatibility with the cloud storage is ensured.

Section 3.5.5 examines the cost of running workloads and storing their results on-premises and in the cloud, and we found that when the cluster is heavily used, the cost of computing power in the cloud is prohibitive, this is without considering other expenses like traffic and storage.

When considering the storage technologies, and from a pure cost point of view, the cost of using cloud storage for long term archiving of vast amounts of HPC data, as demonstrated by eq. (3.8) leads to an increase in the total storage cost, which makes it unsuitable for applications that produce large amounts of results data, for example, climate modeling. Furthermore, since simulation results are evaluated over many years, long-term storability is a fundamental prerequisite that will be very expensive when using an on-demand storage model. Other applications like HTC [1] applications, which load the input data from internet archives and require large amounts of computing, or applications that just do not need massive input data and do not produce lots of data, can cost-effectively use a cloud storage archive; examples include protein folding, financial modeling, and earthquake simulation.

On the other hand, section 3.5.5 shows that substituting the shared parallel file system with a cloud on-demand filesystem is a cost-effective endeavor. The performance feasibility of such a solution is thoroughly investigated in chapter 4 and chapter 5.

To answer the rest of the question *which I/O interfaces are suitable?*, we need to compare HPC-specific interfaces like MPI or other POSIX compliant interfaces to an object storage interface like S3 using REST/HTTP as a transport protocol.

Thus, in chapter 4, a first assessment of using REST as a storage protocol in an HPC environment is provided.

A performance model for the relevant HTTP Get/Put operation based on hardware counters is presented and experimentally validated. The obtained results reveal that an accurately configured REST implementation can provide high performance and match HPC-specific implementation of MPI in terms of throughput for most file sizes and in terms of latency for file sizes exceeding one MB.

By considering the CPU and Memory cost introduced by the data movement, the developed model reasonably covered the different protocols' general behavior and confirmed the expected behavior.

The new techniques introduced in the more recent versions of HTTP (using a small number of connections, multiplexing the HTTP datagram, compressing the header, and allowing the server to "push" data pro-actively to the client while eventually using UDP to accomplish these) carry the potential of improving performance and, thus, providing a perspective for using cloud storage within HPC environments.

[1] High-Throughput Computing

In chapter 5, the performance of the S3 interface offered by different object storage implementations on HPC and in the cloud is thoroughly investigated.

The results indicate that S3 implementations such as MinIO are not yet ready to serve HPC workloads directly, mainly because of the drastic performance loss and the lack of scalability.

☐ Is moving I/O demanding applications from on-premises to the public cloud a cost-effective solution compared to a hybrid alternative?

■ In section 3.5 the cost efficiency of moving applications from on-premises to the cloud and vice-versa is thoroughly investigated, the equations needed to assess the cost of resources, whether on-premises or in the cloud, are provided. Equation (3.7) depicts all the factors that should be considered when running workloads in the cloud and proves that long-running compute-intensive workloads can generate considerable costs in the cloud; the case study found in section 3.5.5 also confirms this statement.

I/O intensive applications also incur high costs in the cloud since providing a certain baseline of provisioned IOPS[2] is another cost factor that should be considered in eq. (3.7); on the other hand, a hybrid scenario where vast amounts of data are transferred between HPC and Cloud can also be time-consuming and costly due to the traffic costs that should be added to the equation.

Other questions from section 1.3.1 are also addressed:

☐ To what extent can we use cloud storage in an HPC environment, what overhead to expect and how can we minimize this overhead?

■ Chapter 5 addresses this question and provides the benchmark results when comparing different S3 implementations to Lustre, one of the most commonly used parallel filesystems within an HPC environment.

The results indicate that S3 implementations such as MinIO are not yet ready to serve HPC workloads directly, mainly because of the drastic performance loss and the lack of scalability.

We found that the remote access to S3 is mainly responsible for the performance loss and should be addressed. Using a load balancer in front of the S3 server nodes not only represents a single point of failure but is also a bottleneck for the data transfer. We used the already existing parallel shared filesystem to leverage an S3 capable application by introducing the local gateway mode. With the advent of filesystem capable of using the node-local storage (ramdisk, SSD, NVRAM) such as GekkofsVef et al., 2020, the local-gw mode can also be extended to accommodate such underlying filesystems.

☐ How can we achieve optimal data sharing between HPC and cloud resources?

■ This question implies that the data is being shared between on-premises and in the cloud resources, which denotes the use of a hybrid solution, for example, a cloud-bursting solution. As seen in section 2.7.1 or fig. 3.6, even though this solution shows an acceptable compute overhead, the network and I/O components present a considerable overhead since the network channels will cause significant data-out charges and add latency to the application. Furthermore, data synchronization will strain the network and storage resources both

[2] `https://aws.amazon.com/ebs/pricing/`

on-premises and in the cloud. Not to mention the considerable administrative effort and the traffic costs introduced by this approach.

Eventually, this overhead depends on the size of the data to be shared. Still, since HPC and Big Data applications typically handle or produce large amounts of data, we should thoroughly investigate the reasons behind sharing such massive data.

Supposing that the data produced by a particular HPC application is to be analyzed by a Big-Data application running in the cloud, the trivial approach illustrated in fig. 7.1 is to move the results data to some cloud storage and then launch the big-data application against this dataset. A connector is needed to interact with the cloud storage; we can achieve this by using some type of cloud storage like the S3A[3] client, which offers high-performance I/O against Amazon S3 object store and compatible implementations and is actively maintained by the Hadoop project itself.

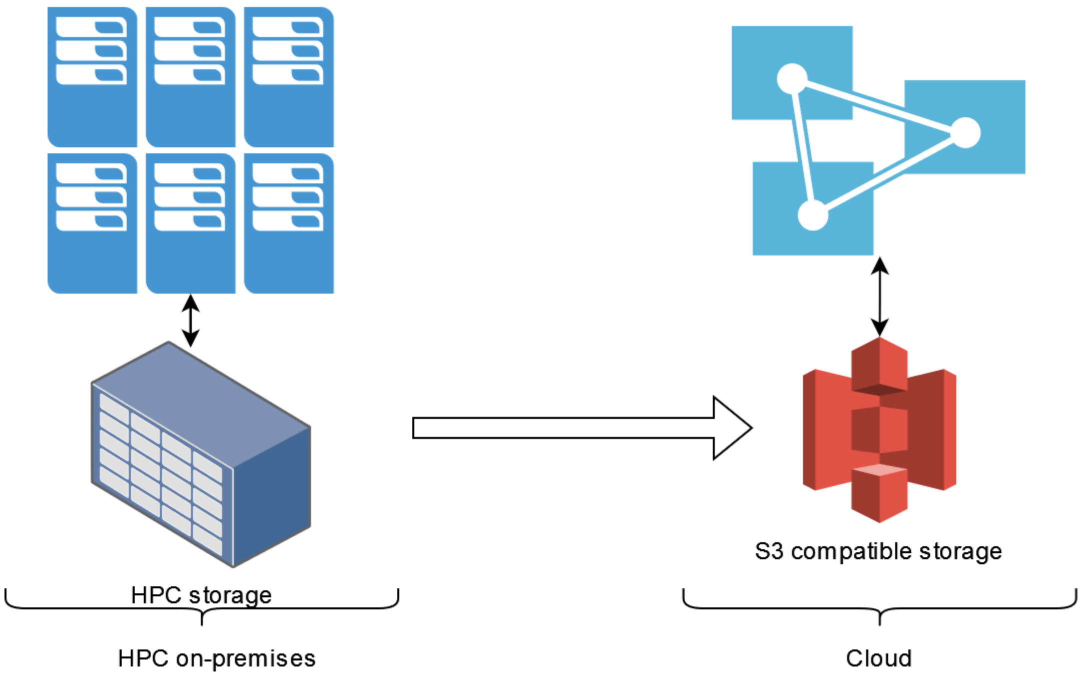


FIGURE 7.1: Data sharing in a converged system

However, this approach contradicts the principle of data locality, which consists of moving the computation close to where the actual data resides instead of moving large data to computation to minimize network congestion and increase the overall throughput of the system[4].

Building upon this principle, the optimal solution is using an HPC Cloud converged system capable of running both types of workload with minimal "code transformation," in the sense that the application code should undergo a minimal or ideally no modification to run on the converged system.

The HPC application generating the data can still run on HPC on-premises or even in an HPC Cloud environment, if applicable, and the data analytics

[3]https://hadoop.apache.org/docs/r3.0.3/hadoop-aws/tools/hadoop-aws/index.html
[4]https://techvidvan.com/tutorials/data-locality-in-hadoop-mapreduce

application can use a wrapper library, similar to S3Embedded, to communicate seamlessly and efficiently with the same storage, as depicted by fig. 7.2.

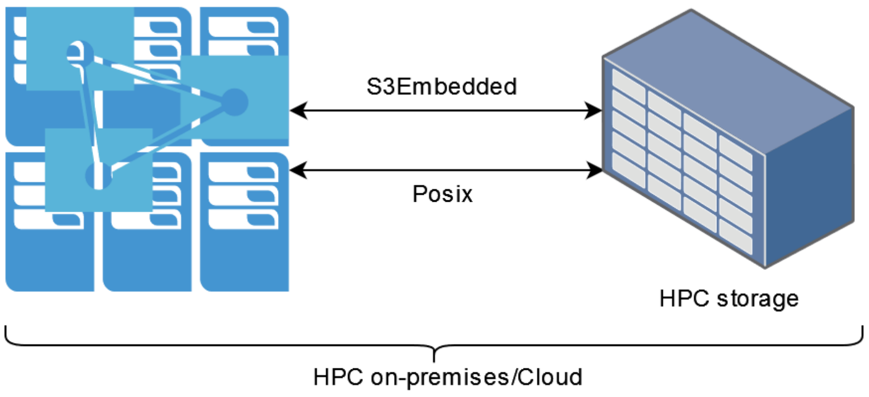


FIGURE 7.2: Data sharing in a converged system

In fig. 7.2 S3Embedded can refer to any embedded library capable of converting existing S3 application storage calls to their HPC storage equivalents.

The S3Embedded library presented in chapter 6 is considered as a proof of concept, developed using the C language and thoroughly tested for performance and scalability using the IO500 benchmark: it can seamlessly convert S3 calls to POSIX calls and can be dynamically linked to any application using the LibS3 library to interact with S3 compatible storage.

As explained in chapter 6, it would be beneficial for S3Embedded to use the native API provided by a performant object store to avoid the overhead of a full-featured POSIX file system. In this case, the underlying file system does not have to perform redundant procedures such as path lookup and permission checking.

S3Embedded is a vendor-agnostic abstraction layer that can be carefully placed between the available storage in the converged ecosystem, and the S3 capable application accessing the storage clustre. It leverages an storage abstraction layer for multiple platforms and enables the portability of different applicatons even if the data itself is not as portable due to either size or cost. Our performance results show that this layer imposes a small overhead for typical data-object operations and a reasonable overhead for metadata-intensive operations. S3Embedded allows an application to be moved with almost no code changes and presents a foundation for platform-agnostic applications which, due to their mobility, can be run on any converged system while optimizing on cost and performance.

It is also possible to implement S3Embedded using other programming languages to act as a drop-in replacement for other commonly used S3 client libraries; some sample implementations might be as follows:

- In python as a wrapper for the boto3 library.
- In node.js as a wrapper for the Knox[5] library.
- In PHP for the amazon-s3-php-class[6] library.

[5]https://github.com/Automattic/knox
[6]https://github.com/tpyo/amazon-s3-php-class

7.1.1 Contributions and Novelty

The contributions and novelty of this work can be summarized as follows:

- Giving a subjective non-biased definition of HPC and Cloud Convergence.
- Providing a fully-fledged assessment model to measure the degree of HPC and Cloud convergence of current and future systems.
- Providing a comprehensive cost study that applies to both HPC and Cloud systems and will help decision-makers choose the best cost-effective solutions when considering moving on-premises workloads into the cloud, or from cloud to on-premises, or even a mix of both solutions.
- Investigating the performance overhead introduced by using cloud storage inside HPC by thoroughly analyzing the performance of REST/HTTP in comparison to HPC specific protocols
- Introducing a performance model based on hardware counters that covers the general behavior of the different protocols.
- Analysing the performance of the S3 interface in HPC and Cloud.
- Amending IO500 to benchmark the S3 interface and broadening the scope of the IO500 usage to enable the community to track the performance growth of S3 over the years and analyze changes in the S3 storage landscape, which will encourage the sharing of best practices for performance optimization.
- Introducing the S3Embedded library to overcome some S3 design problems and performance issues and make it suitable for HPC usage. Its use as a drop-in replacement for LibS3 enables the seamless portability of S3 enabled applications while leveraging HPC standard storage.

7.2 Discussion and Future Work

HPC and Big-Data/Cloud workflows derive from different assumptions that led to different underlying storage architectures and, as such, to various optimization techniques, which seem to be incompatible with one another from an abstract perspective.

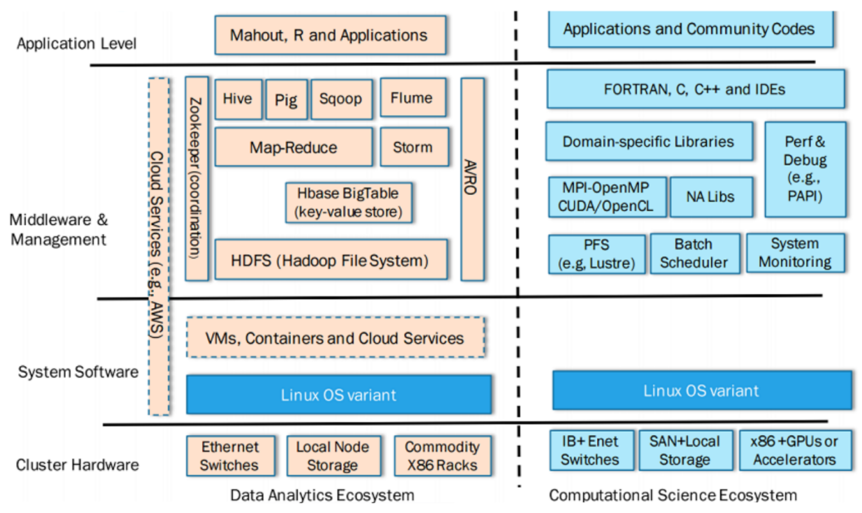


FIGURE 7.3: HPC Ecosystem vs Big-Data Ecosystem, as depicted by (Reed and Dongarra, 2015)

Typical HPC applications are generally associated with modeling and simulation, while Big-Data focuses on the analytics aspect. As seen in fig. 7.3, the difference spans between diverse dimensions starting from the underlying hardware infrastructure to the resource management and job scheduling and including the programming languages used.

However, in recent times, we are increasingly noticing many trends:

- The use of Machine Learning is increasing, not just to interpret results from massive data outputs but also to optimally manage and control computations resources usage, including storage.
- The vast amount of generated data is growing exponentially; it is typical to have petabytes of data generated from just one experiment. As such, data movement is no longer a viable or cost and time-effective option, and the need for real-time data analytics, requiring large-scale computations to be performed closer to the data and data infrastructures, based on the principle of data locality and to adapt to HPC-like modes of operation.
- Data generation is no longer considered the research bottleneck; data management and analysis are now considered the main congestion points.
- Containers and container orchestration platforms like Kubernetes are becoming the one-key solution for all cloud-based software requirements. They have proven to be cost-effective and highly portable for hosting and managing Big Data applications. Kubernetes is substituting other mature Big Data platforms like Hadoop because of its unique features as a flexible and scalable microservice-based architecture.
- Edge computing is also gaining momentum; it refers to the decentralized data processing at the edge of the network, i.e., near the data source. Instead of sending large amounts of data for central processing, only a relevant subset is collected and transmitted, saving bandwidth and storage space.

Hence, it is necessary to overcome these differences and converge on the architectures so that scientific workflows do not need to differentiate between the two ecosystems but benefit from the advantages of both.

This work focuses on the convergence at the storage level and presents the possibility to integrate enormous and diverse workloads on current HPC systems cost-effectively: A highly desired HPC Cloud converged system consists of an HPC System offering a cloud storage interface, with the use of a wrapper library like S3embedded it is possible to leverage existing shared filesystem and even local nodes storage to accommodate both Cloud and HPC applications, without modification.

The need for tools and benchmarks to understand the common issues across HPC simulations, big data, and ML applications is crucial; full-featured I/O benchmarks like the IO500 with the extended S3 interface presented in chapter 5 contribute to fulfilling this need.

Moving forward, the use of specific HPC-oriented containerization technologies like Charliecloud is a good prospect for supporting the portability and deployability of software stacks on different HPC clusters with no significant impact on performance. The proposed converged system can be extended to include this as illustrated in fig. 7.4.

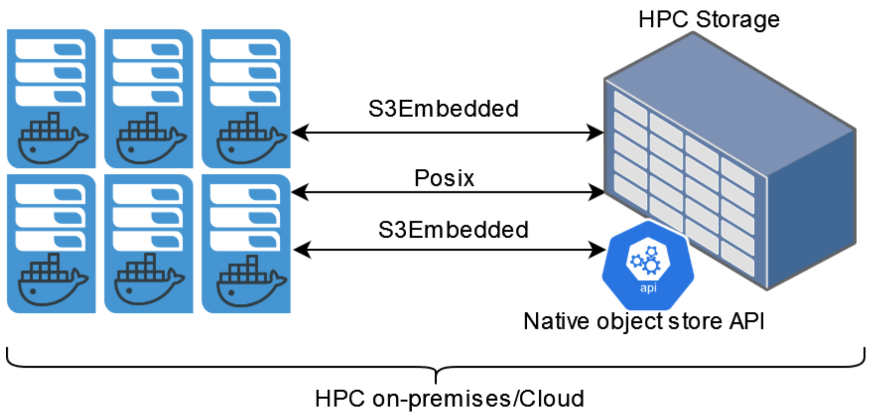


FIGURE 7.4: HPC Cloud coveted converged system

7.2.1 Future Considerations

The computing ecosystems of tomorrow are most likely to be different from the current ones. Future computing will likely include edge, cloud, and high-performance computing combinations.

The hardware heterogeneity, i.e., the use of specialized processors such as GPU[7]s and FPGA[8]s, will more likely increase in future systems as the performance improvements provided by integrated circuit scaling starts to disappear(Dayan et al., 2021).

New programming paradigms and operating and runtime systems will be needed to provide new abstractions and services to make this a seamless ecosystem. Systems will need to be flexible and have low latency at all levels to support new use cases effectively.

This would be favored by more collaboration between the HPC, BD, and ML communities for rapid and efficient progress toward a converged ecosystem that effectively serves all three communities.

Furthermore, standards take a long time and much effort to develop, so instead of introducing new standards, the use of already established and mature standards

[7]Graphics Processing Units
[8]Field Programmable Gate Array

is the best and most cost-effective way to bring forward the scientific community and achieve the business goals of any enterprise.

From a storage perspective, standards like S3 should be fostered to accomplish a seamless and effortless integration.

One of the benefits of utilizing wrapper libraries like S3Embedded is its versatility; no special hardware or software is required to use it fully. This helps avoid vendor lock-in, which is, unfortunately, one of the most common problems in HPC and Cloud landscapes.

Wherever the computing evolution journey may lead, this work provides a full-featured assessment model capable of assessing the convergence level of the future systems and provides the methodology needed to test and benchmark those systems effectively.

Appendix A

Here you will find the tables mentioned in chapter 4:

	Observation		Benchmark Values											Prediction	
size	**t_req**	**N**	**CUCc**	**t_compute_c**	**Pakets**	**CUCs**	**t_compute_s**	**t_compute**	**LLODc**	**t_memc**	**LLODs**	**t_mems**	**t_net**	**t_req_calcul**	***error%***
1	82,3	422480	4,2E+09	4,003	1	9,55E+09	9,080	13,084	6,58E+07	0,871	1,32E+08	1,748	60,000	83,547	1,5
2	81,2	430701	4,1E+09	3,849	1	1,00E+10	9,325	13,174	6,16E+07	0,800	1,38E+08	1,792	60,001	82,965	2,2
4	80,5	433609	4,1E+09	3,817	1	9,90E+09	9,174	12,991	6,47E+07	0,835	1,41E+08	1,816	60,001	83,164	3,4
8	81,3	425321	4,3E+09	4,079	1	9,82E+09	9,275	13,354	6,82E+07	0,897	1,39E+08	1,824	60,003	84,152	3,5
16	84,1	420311	4,2E+09	3,984	1	9,89E+09	9,445	13,430	6,69E+07	0,891	1,35E+08	1,796	60,006	84,144	0,1
32	82,5	425078	4,2E+09	3,941	1	9,98E+09	9,430	13,371	6,18E+07	0,813	1,42E+08	1,866	60,011	83,379	1,0
64	81,5	425079	4,3E+09	4,108	1	9,82E+09	9,274	13,382	6,93E+07	0,912	1,33E+08	1,748	60,022	84,273	3,4
128	80,4	429573	4,0E+09	3,770	1	1,01E+10	9,455	13,225	6,35E+07	0,827	1,45E+08	1,885	60,045	83,421	3,8
256	81,1	428213	4,1E+09	3,888	1	1,00E+10	9,421	13,309	6,35E+07	0,830	1,45E+08	1,889	60,089	83,592	3,1
512	82,1	422911	4,3E+09	4,047	1	9,89E+09	9,396	13,443	6,68E+07	0,883	1,37E+08	1,808	60,178	84,264	2,6
1K	82,7	423718	4,2E+09	3,998	1	1,00E+10	9,513	13,511	6,68E+07	0,882	1,36E+08	1,801	60,356	84,492	2,2
2K	85,2	414253	4,0E+09	3,838	1	9,65E+09	9,356	13,195	6,87E+07	0,928	1,39E+08	1,872	60,712	85,060	0,2
4K	90,2	402021	4,0E+09	4,024	1	9,39E+09	9,384	13,409	6,66E+07	0,927	1,31E+08	1,817	61,425	85,921	4,8
8K	97,8	378427	4,1E+09	4,299	1	8,77E+09	9,305	13,605	7,06E+07	1,044	1,37E+08	2,022	62,849	88,920	9,1
16K	112,4	341573	3,8E+09	4,457	1	8,40E+09	9,878	14,336	7,22E+07	1,183	1,36E+08	2,231	65,699	94,092	16,3
32K	138,7	296928	3,6E+09	4,805	1	7,23E+09	9,780	14,585	7,25E+07	1,367	1,40E+08	2,646	71,398	102,300	26,2
64K	184,8	240278	3,3E+09	5,547	2	6,63E+09	11,086	16,633	7,20E+07	1,677	1,27E+08	2,954	82,795	110,768	40,1
128K	245,5	191903	3,2E+09	6,596	3	5,62E+09	11,759	18,354	7,81E+07	2,277	2,07E+08	6,043	105,590	137,578	44,0
256K	380,4	134535	3,1E+09	9,185	5	4,27E+09	12,733	21,918	6,77E+07	2,816	3,56E+08	14,808	151,181	193,539	49,1
512K	640,0	84782	3,2E+09	15,079	9	3,16E+09	14,982	30,061	6,16E+07	4,067	4,65E+08	30,701	242,361	307,642	51,9
1M	1.110,0	51181	3,1E+09	24,121	17	2,45E+09	19,224	43,345	6,74E+07	7,366	5,18E+08	56,632	424,722	529,032	52,3
2M	1.810,0	32133	3,2E+09	39,384	33	2,25E+09	28,166	67,550	8,05E+07	14,020	5,18E+08	90,227	789,444	951,470	47,4
4M	1.990,0	29662	5,6E+09	75,919	65	3,32E+09	44,891	120,810	1,48E+08	27,873	1,05E+09	198,704	1518,888	1842,691	7,4
8M	3.270,0	18164	5,7E+09	127,011	129	3,00E+09	66,369	193,379	1,66E+08	51,132	8,01E+08	246,788	2977,777	3421,907	4,6
16M	6.330,0	9409	5,5E+09	233,488	257	2,90E+09	123,797	357,285	1,76E+08	104,443	8,22E+08	488,776	5895,553	6745,678	6,6
32M	12.120,0	4945	6,0E+09	489,250	513	3,21E+09	260,634	749,883	1,68E+08	190,351	5,89E+08	666,065	11731,107	13150,766	8,5
64M	27.430,0	2185	7,2E+09	1327,588	1025	2,81E+09	516,996	1844,585	1,65E+08	423,206	5,90E+08	1510,658	23402,214	26761,585	2,4
128M	54.120,0	1108	7,4E+09	2682,650	2049	2,76E+09	1001,701	3684,351	1,64E+08	829,011	6,57E+08	3318,184	46744,427	53751,008	0,7

TABLE A.1: Predictive model error rate for the REST Results

	Observation	Benchmark Values										Prediction	
size	**t_req**	**CUCc**	**t_compute_c**	**CUCs**	**t_compute_s**	**t_compute**	**LLODc**	**t_memc**	**LLODs**	**t_mems**	**t_net**	**t_req_calcul**	***error%***
1	30,5	296028444	11,771	291353045	11,585	23,356	1229246	0,681	1206932	0,669	6,000	30,706	0,7
2	30,4	327431171	13,020	292704984	11,639	24,658	1175785	0,652	1166305	0,646	6,001	31,957	5,1
4	30,5	300890508	11,964	274284908	10,906	22,871	1172410	0,650	1168322	0,647	6,001	30,169	1,0
8	30,5	404896949	16,100	305377217	12,143	28,243	1182522	0,655	1178837	0,653	6,003	35,554	16,5
16	30,8	316997746	12,605	287849223	11,446	24,051	1180714	0,654	1172836	0,650	6,006	31,360	1,7
32	31,2	286575950	11,395	277716698	11,043	22,438	1187950	0,658	1167981	0,647	6,012	29,755	4,8
64	30,6	316161462	12,572	275852802	10,969	23,540	1183257	0,656	1176745	0,652	6,023	30,871	1,0
128	30,4	316664614	12,592	288452527	11,470	24,061	1191044	0,660	1186700	0,658	6,047	31,425	3,4
256	30,4	288921716	11,488	257960174	10,257	21,746	1133372	0,628	1266736	0,702	6,093	29,169	4,1
512	30,6	263360043	10,472	266269134	10,588	21,060	1143645	0,634	1290834	0,715	6,186	28,595	6,6
1K	31,2	264238769	10,507	341907488	13,595	24,102	1140616	0,632	1329922	0,737	6,372	31,844	2,2
2K	31,6	270848151	10,770	269196813	10,704	21,474	1144686	0,634	1276811	0,707	6,745	29,560	6,6
4K	41,5	299244392	11,899	305548716	12,150	24,048	1187661	0,658	1348270	0,747	7,489	32,943	20,7
8K	43,4	337268043	13,411	282832499	11,246	24,657	1196629	0,663	1412176	0,782	8,979	35,082	19,1
16K	44,7	279020149	11,095	252083024	10,024	21,118	1126909	0,624	1210873	0,671	11,958	34,371	23,1
32K	59,9	299052988	11,891	286286819	11,384	23,275	1269937	0,704	1475545	0,818	17,916	42,712	28,6
64K	107,7	338639395	13,465	280372917	11,148	24,614	1304406	0,723	1503829	0,833	29,831	56,001	48,0
128K	133,5	403295684	16,036	290946244	11,569	27,605	1536219	0,851	1582617	0,877	53,663	82,996	37,8
256K	198,1	558458766	22,206	349265302	13,888	36,094	1960144	1,086	1828675	1,013	101,325	139,518	29,6
512K	330,5	928192083	36,908	444074939	17,658	54,565	2949189	1,634	2743634	1,520	196,650	254,370	23,0
1M	590,3	1687469047	67,099	584198855	23,230	90,328	4560377	2,527	4266983	2,364	387,300	482,520	18,3
2M	1107,9	3133111681	124,582	950323406	37,788	162,370	7837176	4,343	7142188	3,958	768,601	939,271	15,2
4M	2194,3	6024059749	239,535	1575293378	62,638	302,173	14162018	7,847	13039777	7,225	1531,201	1848,447	15,8
8M	4347,1	11616099576	461,891	2810447870	111,752	573,643	27912105	15,466	24215159	13,418	3056,403	3658,930	15,8
16M	8745,6	23333658230	927,817	5544310402	220,458	1148,275	55868947	30,957	47987477	26,590	6106,806	7312,628	16,4

TABLE A.2: Predictive model error rate for the MPIoTCP Results

size	t_req	N	60s/N	CUC C	t_compute_c	Pakete	CUCs	t_compute_s	t_compute	LLODc	t_memc	LLODs	t_mems	t_net	t_req_calcul	error%
1	82.3	422480	142.019	4.2E+09	4.003	1	9.55E+09	9.080	13.084	6.58E+07	0.871	1.32E+08	1.748	60.000	75.704	8.0
2	81.2	430701	139.308	4.1E+09	3.849	1	1.00E+10	9.325	13.174	6.16E+07	0.800	1.38E+08	1.792	60.001	75.766	6.7
4	80.5	433609	138.374	4.1E+09	3.817	1	9.90E+09	9.174	12.991	6.47E+07	0.835	1.41E+08	1.816	60.001	75.644	6.0
8	81.3	425321	141.070	4.3E+09	4.079	1	9.82E+09	9.275	13.354	6.82E+07	0.897	1.39E+08	1.824	60.003	76.077	6.5
16	84.1	420311	142.751	4.2E+09	3.984	1	9.89E+09	9.445	13.430	6.69E+07	0.891	1.35E+08	1.796	60.006	76.123	9.4
32	82.5	425078	141.151	4.2E+09	3.941	1	9.98E+09	9.430	13.371	6.18E+07	0.813	1.42E+08	1.866	60.011	76.062	7.8
64	81.5	425079	141.150	4.3E+09	4.108	1	9.82E+09	9.274	13.382	6.93E+07	0.912	1.33E+08	1.748	60.022	76.065	6.7
128	80.4	429573	139.674	4.0E+09	3.770	1	1.01E+10	9.455	13.225	6.35E+07	0.827	1.45E+08	1.885	60.045	75.982	5.5
256	81.1	428213	140.117	4.1E+09	3.888	1	1.00E+10	9.421	13.309	6.35E+07	0.830	1.45E+08	1.889	60.089	76.118	6.1
512	82.1	422911	141.874	4.3E+09	4.047	1	9.89E+09	9.396	13.443	6.68E+07	0.883	1.37E+08	1.808	60.178	76.313	7.1
1024	82.7	423718	141.604	4.2E+09	3.998	1	1.00E+10	9.513	13.511	6.68E+07	0.882	1.36E+08	1.801	60.356	76.551	7.4
2048	85.2	414253	144.839	4.0E+09	3.838	1	9.65E+09	9.356	13.195	6.87E+07	0.928	1.39E+08	1.872	60.712	76.707	10.0
4096	90.2	402021	149.246	4.0E+09	4.024	1	9.39E+09	9.384	13.409	6.66E+07	0.927	1.31E+08	1.817	61.425	77.578	14.0
8192	97.8	378427	158.551	4.1E+09	4.299	1	8.77E+09	9.305	13.605	7.06E+07	1.044	1.37E+08	2.022	62.849	79.521	18.7
16384	112.4	341573	175.658	3.8E+09	4.457	1	8.40E+09	9.878	14.336	7.22E+07	1.183	1.36E+08	2.231	65.699	83.449	25.8
32768	138.7	296928	202.069	3.6E+09	4.805	1	7.23E+09	9.780	14.585	7.25E+07	1.367	1.40E+08	2.646	71.398	89.996	35.1
65536	184.8	240278	249.711	3.3E+09	5.547	2	6.63E+09	11.086	16.633	7.20E+07	1.677	1.27E+08	2.954	82.795	104.059	43.7
131072	245.5	191903	312.658	3.2E+09	6.596	3	5.62E+09	11.759	18.354	7.81E+07	2.277	2.07E+08	6.043	105.590	132.265	46.1
262144	380.4	134535	445.981	3.1E+09	9.185	5	4.27E+09	12.733	21.918	6.77E+07	2.816	3.56E+08	14.808	151.181	190.722	49.9
524288	640.0	84782	707.697	3.2E+09	15.079	9	3.16E+09	14.982	30.061	6.16E+07	4.067	4.65E+08	30.701	242.361	307.190	52.0
1048576	1,110.0	51181	1172.310	3.1E+09	24.121	17	2.45E+09	19.224	43.345	6.74E+07	7.366	5.18E+08	56.632	424.722	532.066	52.1
2097152	1,810.0	32133	1867.239	3.2E+09	39.384	33	2.25E+09	28.166	67.550	8.05E+07	14.020	5.18E+08	90.227	789.444	961.241	46.9
4194304	1,990.0	29662	2022.790	5.6E+09	75.919	65	3.32E+09	44.891	120.810	1.48E+08	27.873	1.05E+09	198.704	1518.888	1866.276	6.2
8388608	3,270.0	18164	3303.237	5.7E+09	127.011	129	3.00E+09	66.369	193.379	1.66E+08	51.132	8.01E+08	246.788	2977.777	3469.076	6.1
16777216	6,330.0	9409	6376.873	5.5E+09	233.488	257	2.90E+09	123.797	357.285	1.76E+08	104.443	8.22E+08	488.776	5895.553	6846.057	8.2
33554432	12,120.0	4945	12133.468	6.0E+09	489.250	513	3.21E+09	260.634	749.883	1.68E+08	190.351	5.89E+08	666.065	11731.107	13337.407	10.0
67108864	27,430.0	2185	27459.954	7.2E+09	1327.588	1025	2.81E+09	516.996	1844.585	1.65E+08	423.206	5.90E+08	1510.658	23402.214	27180.662	0.9
134217728	54,120.0	1108	54151.625	7.4E+09	2682.650	2049	2.76E+09	1001.701	3684.351	1.64E+08	829.011	6.57E+08	3318.184	46744.427	54575.973	0.8

TABLE A.3: Predictive model error rate for the MPIoRDMA Results

The results of all the benchmarks conducted on the Mistral and WR Cluster, and the scripts needed to reproduce those benchmarks were published alongside the Paper entitled: Investigating the Overhead of the REST Protocol when Using Cloud Services for HPC Storage [1] and are found online at `https://github.com/http-3/rest-overhead-paper`

[1]`https://doi.org/10.1007/978-3-030-59851-8_10`

Appendix B

Glossary

ACL	Access Control List
API	Application Programming Interface
AWS	Amazon Web Services
CUC	CPU_CLK_UNHALTED_CORE
DKRZ	Deutsches Klimarechenzentrum: German Climate Computing Center
DPC	Distributed Parallel Client
FPGA	Field Programmable Gate Array
GCS	Google Cloud Storage
GPU	Graphics Processing Units
GWDG	Gesellschaft für Wissenschaftliche Datenverarbeitung mbH Göttingen
HPC	High Performance Computing
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
LMMA	Logging, Metrics, Monitoring, and Alerting
MPI	Message Passing Interface
MPU	S3 MultiPart Upload
MSS	Maximum Segment Size
MTU	Maximum Transfer Unit
MTU	Maximum Transmission Unit
PaaS	Platform as a Service
PFS	Parallel File System
PPN	Process per Node
RDMA	Remote Direct Memory Access
REST	REpresentational State Transfer
RTT	Round Trip Time

S3	Simple Storage Service
SaaS	Software as a Service
SDK	Software Development Kit
SQL	Simple Structured Query Language
TTL	Time to Live
UDP	User Datagram Protocol

Appendix C

List of Publications resulting from this Dissertation

C.1 Publications with peer review process

- Gadban, Frank and Julian Kunkel (2021). “Analyzing the Performance of the S3 Object Storage API for HPC Workloads”. In: Applied Sciences 11.18, p. 8540. URL:https://doi.org/10.3390/app11188540

  Author Contributions
  Conceptualization, F.G. and J.K.; methodology, F.G. and J.K.; software, F.G. and J.K.; validation, F.G. and J.K.; formal analysis, F.G. and J.K.; investigation, F.G.; resources, F.G.; data curation, F.G. and J.K.; writing—original draft preparation, F.G.; writing—review and editing, F.G. and J.K.; visualization, F.G. and J.K.; supervision, J.K.; project administration, J.K.

- Gadban, Frank, Julian Kunkel, and Thomas Ludwig (2020). “Investigating the Overhead of the REST Protocol When Using Cloud Services for HPC Storage”. In: International Conference on High Performance Computing. Springer, pp. 161–176. URL:https://doi.org/10.1007/978-3-030-59851-8_10

  Author Contributions
  Conceptualization, F.G. and J.K.; methodology, F.G. and J.K.; software, F.G.; validation, F.G. and J.K.; formal analysis, F.G. and J.K.; investigation, F.G.; resources, F.G. and T.L.; data curation, F.G. and J.K.; writing—original draft preparation, F.G.; writing—review and editing, F.G. and J.K.; visualization, F.G. and J.K.; supervision, T.L. and J.K.; project administration, J.K.

Bibliography

Abraham, Erika et al. (2015). “Preparing HPC applications for exascale: Challenges and recommendations”. In: *2015 18th International Conference on Network-Based Information Systems*. IEEE, pp. 401–406.

Abraham, Subil et al. (2020). “On the use of containers in high performance computing environments”. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, pp. 284–293.

Anderson, David P (2004). “Boinc: A system for public-resource computing and storage”. In: *Fifth IEEE/ACM international workshop on grid computing*. IEEE, pp. 4–10.

— (2020). “BOINC: a platform for volunteer computing”. In: *Journal of Grid Computing* 18.1, pp. 99–122.

Arsuaga-Ríos, María et al. (2015). “Using S3 cloud storage with ROOT and CvmFS”. In: *Journal of Physics: Conference Series*. Vol. 664. 2. IOP Publishing, p. 022001.

Association, Infiniband Trade (2020). *About Infiniband*. https://www.infinibandta.org/about-infiniband/. [Online; accessed 29-July-2019].

AWS (n.d.[a]). *AWS*. https://aws.amazon.com/hpc/. [Online; accessed 19-July-2019].

— (n.d.[b]). *Multipart upload overview*. https://docs.aws.amazon.com/AmazonS3/latest/dev/mpuoverview.html. [Online; accessed 19-July-2020].

— (n.d.[c]). *Performance Design Patterns for Amazon S3*. https://docs.aws.amazon.com/AmazonS3/latest/dev/optimizing-performance-design-patterns.html. [Online; accessed 19-sep-2020].

— (2020). *AWS S3*. https://aws.amazon.com/de/s3/. [Online; accessed 19-July-2020].

Barkes, Jason et al. (1998). “GPFS: a parallel file system”. In: *IBM International Technical Support Organization*.

Bent, John et al. (2009). “PLFS: a checkpoint filesystem for parallel applications”. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, p. 21.

Bessani, Alysson et al. (2013). “DepSky: dependable and secure storage in a cloud-of-clouds”. In: *Acm transactions on storage (tos)* 9.4, pp. 1–33.

bji (n.d.). *libs3 removes support for signature V2*. https://github.com/bji/libs3/pull/50. [Online; accessed 19-Aug-2020].

Bjornson, Zach (2015). *Cloud Storage Performance*. https://blog.zachbjornson.com/2015/12/29/cloud-storage-performance.html. [Online; accessed 19-July-2020].

Bortolotti, Daniela et al. (2011). “Comparison of UDP Transmission Performance Between IP-Over-InfiniBand and 10-Gigabit Ethernet”. In: *IEEE Transactions on Nuclear Science* 58.4, pp. 1606–1612.

Borzemski, Leszek and Gabriel Starczewski (2009). “Application of transfer regression to TCP throughput prediction”. In: *2009 First Asian Conference on Intelligent Information and Database Systems*. IEEE, pp. 28–33.

Braam, Peter (2019). “The Lustre storage architecture”. In: *arXiv preprint arXiv:1903.01955*.

Bruno, Greg et al. (2004). "Rolls: Modifying a standard system installer to support user-customizable cluster frontend appliances". In: *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*. IEEE, pp. 421–430.

Bundesanzeiger (2020). *DKRZ Jahresabschluss*. `https://www.bundesanzeiger.de`. [Online; accessed 19-July-2021].

Chang, Chen-Shang and Joy A. Thomas (1995). "Effective bandwidth in high-speed digital networks". In: *IEEE Journal on Selected areas in Communications* 13.6, pp. 1091–1100.

Chen, Donglin et al. (2020). "Characterizing scalability of sparse matrix–vector multiplications on phytium ft-2000+". In: *International Journal of Parallel Programming* 48.1, pp. 80–97.

Chen, Shuang et al. (2017). "Workload characterization of interactive cloud services on big and small server platforms". In: *Workload Characterization (IISWC), 2017 IEEE International Symposium on*. IEEE, pp. 125–134.

Cinquini, Luca et al. (2014). "The Earth System Grid Federation: An open infrastructure for access to distributed geospatial data". In: *Future Generation Computer Systems* 36, pp. 400–417.

Cloudflare (2020). *Implementation of the QUIC protocol*. `https://github.com/cloudflare/quiche`. [Online; accessed 01-April-2020].

Corbett, Peter et al. (1996). "Overview of the MPI-IO parallel I/O interface". In: *Input/Output in Parallel and Distributed Computer Systems*. Springer, pp. 127–146.

Coym, Johannes (2021). "Analysis of Elastic Cloud Solutions in an HPC Environment". In.

Dayan, Niv et al. (2021). "The end of Moore's law and the rise of the data processor". In: *Proceedings of the VLDB Endowment* 14.12, pp. 2932–2944.

Denis, Alexandre and François Trahay (2016). "MPI overlap: Benchmark and analysis". In: *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, pp. 258–267.

Devresse, Adrien and Fabrizio Furano (2014). "Efficient HTTP based I/O on very large datasets for high performance computing with the libdavix library". In: *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, pp. 194–205.

DKRZ (2020). *Mistral*. `https://www.dkrz.de/up/systems/mistral/configuration`. [Online; accessed 19-July-2020].

Dumazet, Eric (2012). *Increase loopback mtu*. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=0cf833aefaa85bbfce3ff70485e5534e09254`
[Online; accessed 24-Feb-2020].

Emeras, Joseph et al. (2016). "HPC or the Cloud: a cost study over an XDEM Simulation". In: *Proc. of the 7th International Supercomputing Conference in Mexico (ISUM 2016)*. Puebla, México.

Expósito, Roberto R et al. (2013). "Performance analysis of HPC applications in the cloud". In: *Future Generation Computer Systems* 29.1, pp. 218–229.

Farber, Rob (2011). *CUDA application design and development*. Elsevier.

Felter, Wes et al. (2015). "An updated performance comparison of virtual machines and linux containers". In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, pp. 171–172.

Fielding, Roy T (2008). "REST APIs must be hypertext-driven". In: *Untangled musings of Roy T. Fielding*, p. 24.

Folk, Mike et al. (2011). "An overview of the HDF5 technology suite and its applications". In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. ACM, pp. 36–47.

Forum, MPI (n.d.). *MPI2 Report*. https://www.mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf. [Online; accessed 29-July-2021].

Gadban, Frank and Julian Kunkel (2021). "Analyzing the Performance of the S3 Object Storage API for HPC Workloads". In: *Applied Sciences* 11.18, p. 8540. URL: https://doi.org/10.3390/app11188540.

Gadban, Frank, Julian Kunkel, and Thomas Ludwig (2020). "Investigating the Overhead of the REST Protocol When Using Cloud Services for HPC Storage". In: *International Conference on High Performance Computing*. Springer, pp. 161–176. URL: https://doi.org/10.1007/978-3-030-59851-8_10.

Garfinkel, Simson (2007). "An evaluation of Amazon's grid computing services: EC2, S3, and SQS". In.

Gettys, Jim (1998). *SMUX Protocol Specification*. https://www.w3.org/TR/1998/WD-mux-19980710. [Online; accessed 19-July-2019].

Glozer, Will (2020). *wrk - a HTTP benchmarking tool*. https://github.com/wg/wrk. [Online; accessed 19-July-2020].

Goodell, David et al. (2012). "An evolutionary path to object storage access". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, pp. 36–41.

Graham, Richard L, Timothy S Woodall, and Jeffrey M Squyres (2005). "Open MPI: A flexible high performance MPI". In: *International Conference on Parallel Processing and Applied Mathematics*. Springer, pp. 228–239.

Grant, Ryan E, Pavan Balaji, and Ahmad Afsahi (2010). "A study of hardware assisted IP over InfiniBand and its impact on enterprise data center performance". In: *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, pp. 144–153.

Greguska, Frank (2018). "Using S3 in Apache Science Data Analytics Platform". In.

Gropp, William (2016). *Introduction to MPI I/O*. http://wgropp.cs.illinois.edu/courses/cs598-s16/lectures/lecture32.pdf. [Online].

Gruber, Thomas (2020). *Likwid:about L3 evict*. https://github.com/RRZE-HPC/likwid/issues/213. [Online; accessed 13-July-2020].

Gutierrez, Careres and Franco Jesus (2020). *Towards an S3-based, DataNode-lessimplementation of HDFS*.

h2load (2020). *benchmarking tool for HTTP/2 server*. https://nghttp2.org/documentation/h2load.1.html. [Online; accessed 19-October-2020].

HDF-Group (n.d.). *HSDS REST-based service for HDF5 data*. https://github.com/HDFGroup/hsds. [Online; accessed 19-July-2020].

He, Qi, Constantinos Dovrolis, and Mostafa Ammar (2007). "On the predictability of large transfer TCP throughput". In: *Computer Networks* 51.14, pp. 3959–3977.

Heichler, Jan (2014). *An introduction to BeeGFS*.

IBM (n.d.). *IBM Cloud*. https://www.ibm.com/cloud/hpc. [Online; accessed 19-July-2019].

IETF (2011). *Request for Comments: 6298*. https://tools.ietf.org/html/rfc6298. [Online; accessed 19-January-2020].

— (2020). *QUIC Working Group*. https://quicwg.org/. [Online; accessed 01-April-2020].

Inc, Google (n.d.). *PerfKit Benchmarker*. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker. [Online; accessed 19-July-2020].

Intel (2009). *An Introduction to the Intel® QuickPath Interconnect*. https://www.intel.com/technology/quickpath/introduction.pdf. [Online; accessed 15-September-2019].

Intel (2014). *Intel® Xeon® Processor E5-2680*. https://ark.intel.com/content/www/us/en/ark/products/81908/intel-xeon-processor-e5-2680-v3-30m-cache-2-50-ghz.html. [Online; accessed 15-September-2019].

— (2020). *Adress Translation on Intel X56xx*. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/277182. [Online; accessed 15-September-2020].

Kelly, Diane F (2007). "A software chasm: Software engineering and scientific computing". In: *IEEE software* 24.6, pp. 120–119.

Kimpe, Dries and Robert Ross (2014). "Storage models: Past, present, and future". In: *High Performance Parallel I/O*, pp. 335–345.

Kneschke, Jan (2020). *lighttpd*. https://www.lighttpd.net/. [Online; accessed 29-July-2020].

Ko, Ryan KL et al. (2012). "Overcoming large data transfer bottlenecks in restful service orchestrations". In: *2012 IEEE 19th International Conference on Web Services*. IEEE, pp. 654–656.

Koch, Fernando, Marcos D Assunçao, and Marco AS Netto (2012). "A cost analysis of cloud computing for education". In: *International Conference on Grid Economics and Business Models*. Springer, pp. 182–196.

Korolev, Vlad (n.d.). *AWS4C - A C library to interface with Amazon Web Services*. https://github.com/vladistan/aws4c. [Online; accessed 19-Aug-2020].

Krishnan, Sriram, Mahidhar Tatineni, and Chaitanya Baru (2011). "myHadoop-Hadoop-on-Demand on traditional HPC resources". In: *San Diego Supercomputer Center Technical Report TR-2011-2, University of California, San Diego*, p. 9.

Kunkel, Julian, Gerald Fredrick Lofstead, and John Bent (2017). *The Virtual Institute for I/O and the IO-500.* Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

Kunkel, Julian Martin and George S Markomanolis (2018). "Understanding metadata latency with MDWorkbench". In: *International Conference on High Performance Computing*. Springer, pp. 75–88.

Kurtzer, Gregory M, Vanessa Sochat, and Michael W Bauer (2017). "Singularity: Scientific containers for mobility of compute". In: *PloS one* 12.5, e0177459.

Lafayette, Lev (2018). "Exploring Issues in Event-Based HPC Cloudbursting". In.

Linthicum, David S (2016). "Emerging hybrid cloud patterns". In: *IEEE Cloud Computing* 3.1, pp. 88–91.

Liu, Jialin et al. (2018). "Evaluation of HPC application I/O on object storage systems". In: *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*. IEEE, pp. 24–34.

Liu, Jiuxing et al. (2004). "Microbenchmark performance comparison of high-speed cluster interconnects". In: *Ieee Micro* 24.1, pp. 42–51.

LLNL (n.d.). *IOR parallel I/O benchmarks*. https://github.com/hpc/ior. [Online; accessed 19-sep-2020].

Lofstead, Jay et al. (2016). "DAOS and friends: a proposal for an exascale storage system". In: *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 585–596.

Lofstead, Jay F et al. (2008). "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)". In: *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pp. 15–24.

Lüttgau, Jakob and Julian Kunkel (2018). "Cost and performance modeling for Earth system data management and beyond". In: *International Conference on High Performance Computing*. Springer, pp. 23–35.

Lüttgau, Jakob et al. (2018). “Survey of storage systems for high-performance computing”. In: *Supercomputing Frontiers and Innovations* 5.1.

Ma, Dan, Wei Zhang, and Qinghua Li (2004). “Dynamic scheduling algorithm for parallel real-time jobs in heterogeneous system”. In: *Computer and Information Technology, 2004. CIT'04. The Fourth International Conference on*. IEEE, pp. 462–466.

Massie, Matt et al. (2013). “Adam: Genomics formats and processing patterns for cloud scale computing”. In: *University of California, Berkeley Technical Report, No. UCB/EECS-2013* 207, p. 2013.

Matri, Pierre et al. (2017). “Could blobs fuel storage-based convergence between HPC and big data?” In: *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, pp. 81–86.

McGough, A Stephen et al. (2014). “Comparison of a cost-effective virtual cloud cluster with an existing campus cluster”. In: *Future Generation Computer Systems* 41, pp. 65–78.

Mell, Peter, Tim Grance, et al. (2011). “The NIST definition of cloud computing”. In.

Merzky, Andre, Ole Weidner, and Shantenu Jha (2015). “SAGA: A standardized access layer to heterogeneous distributed computing infrastructure”. In: *SoftwareX* 1, pp. 3–8.

MinIO-Inc. (n.d.). *Kubernetes Native,High Performance Object Storage*. `https://min.io`. [Online; accessed 19-sep-2020].

Moody, William Clay et al. (2013). “Jummp: Job uninterrupted maneuverable mapreduce platform”. In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, pp. 1–8.

Netto, Marco AS et al. (2018). “HPC cloud for scientific and business applications: taxonomy, vision, and research challenges”. In: *ACM Computing Surveys (CSUR)* 51.1, pp. 1–29.

ngtcp2 (2020). *Effort to implement IETF QUIC protocol*. `https://github.com/ngtcp2/ngtcp2`. [Online; accessed 01-April-2020].

NLANR/DAST (2020). *Iperf*. `https://github.com/esnet/iperf`. [Online; accessed 11-July-2019].

OpenLiteSpeed (2020). *OpenLiteSpeed Web Server*. `https://openlitespeed.org/`. [Online; accessed 19-December-2020].

OpenSSL (2020). *QUIC and OpenSSL*. `https://www.openssl.org/blog/blog/2020/02/17/QUIC-and-OpenSSL/`. [Online; accessed 01-April-2020].

OpenStack-Foundation (n.d.). *OpenStack Swift*. `https://github.com/openstack/swift`. [Online; accessed 19-sep-2020].

Palankar, Mayur R et al. (2008). “Amazon S3 for science grids: a viable solution?” In: *Proceedings of the 2008 international workshop on Data-aware distributed computing*, pp. 55–64.

Palyart, Marc et al. (2012). “HPCML: a modeling language dedicated to high-performance scientific computing”. In: *Proceedings of the 1st International Workshop on Model-Driven Engineering for High Performance and CLoud computing*, pp. 1–6.

Pande, Vijay et al. (2010). “Folding@ home”. In: *Distributed Computing*.

Parashar, Manish et al. (2013). “Cloud paradigms and practices for computational and data-enabled science and engineering”. In: *Computing in Science & Engineering* 15.4, pp. 10–18.

Perna, Gianluca et al. (2022). “A first look at HTTP/3 adoption and performance”. In: *Computer Communications* 187, pp. 115–124.

Persico, Valerio, Antonio Montieri, and Antonio Pescape (2016). “On the network performance of amazon s3 cloud-storage service”. In: *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*. IEEE, pp. 113–118.

Persico, Valerio et al. (2016). "A first look at public-cloud inter-datacenter network performance". In: *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, pp. 1–7.

Priedhorsky, Reid and Tim Randles (2017). "Charliecloud: Unprivileged containers for user-defined software stacks in hpc". In: *Proceedings of the international conference for high performance computing, networking, storage and analysis*, pp. 1–10.

Reed, Daniel A and Jack Dongarra (2015). "Exascale computing and big data". In: *Communications of the ACM* 58.7, pp. 56–68.

Rew, Russ and Glenn Davis (1990a). "NetCDF: an interface for scientific data access". In: *IEEE computer graphics and applications* 10.4, pp. 76–82.

— (1990b). "NetCDF: an interface for scientific data access". In: *IEEE computer graphics and applications* 10.4, pp. 76–82.

Reyes-Ortiz, Jorge L, Luca Oneto, and Davide Anguita (2015). "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf". In: *Procedia Computer Science* 53, pp. 121–130.

Richardson, Leonard and Sam Ruby (2008). *RESTful web services*. " O'Reilly Media, Inc."

Sabin, Jason Allen et al. (2016). *Techniques for cloud bursting*. US Patent 9,454,406.

Sadooghi, Iman et al. (2015). "Understanding the performance and potential of cloud computing for scientific applications". In: *IEEE Transactions on Cloud Computing* 5.2, pp. 358–371.

Salaria, Shweta et al. (2017). "Evaluation of HPC-Big Data applications using cloud platforms". In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, pp. 1053–1061.

Schulz, Karl W et al. (2016). "Cluster computing with OpenHPC". In.

Scott, Stephen L (2001). "OSCAR and the Beowulf arms race for the" cluster standard"". In: *Proceedings 2001 IEEE International Conference on Cluster Computing*. IEEE, pp. 137–137.

Smith, Preston et al. (2019). "Community Clusters or the Cloud: Continuing cost assessment of on-premises and cloud HPC in Higher Education". In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pp. 1–4.

Sterling, Thomas Lawrence (2002). *Beowulf cluster computing with Linux, Second Edition*. MIT press.

Sysoev, Igor (n.d.). *Nginx*. `https://nginx.org`. [Online; accessed 19-July-2020].

Tanaka, Yoshio et al. (2013). "Building secure and transparent inter-cloud infrastructure for scientific applications". In: *Cloud Computing and Big Data* 23, p. 35.

Tene, Gil (2020). *A constant throughput, correct latency recording variant of wrk*. `https://github.com/giltene/wrk2`. [Online; accessed 11-July-2020].

Thakur, Rajeev, William Gropp, and Ewing Lusk (1999). "Data sieving and collective I/O in ROMIO". In: *Proceedings. Frontiers' 99. Seventh Symposium on the Frontiers of Massively Parallel Computation*. IEEE, pp. 182–189.

The MPI Forum, CORPORATE (1993). "MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: ACM, pp. 878–883. ISBN: 0-8186-4340-4. DOI: `10.1145/169627.169855`. URL: `http://doi.acm.org/10.1145/169627.169855`.

Tianhua, Liu et al. (2008). "The design and implementation of zero-copy for linux". In: *2008 Eighth International Conference on Intelligent Systems Design and Applications*. Vol. 1. IEEE, pp. 121–126.

Todd, Lindsay (2017). *POSIX file system basics*. `https://parallelstorage.com/2017/12/29/posix-file-system-basics/`. [Online].

Torrez, Alfred, Reid Priedhorsky, and Timothy Randles (2020). “HPC container runtime performance overhead: At first order, there is none”. In.

Trangoni, Mario and Matías Cabral (2012). “A comparison of provisioning systems for beowulf clusters”. In: *XVIII Congreso Argentino de Ciencias de la Computación*.

Treibig, Jan, Georg Hager, and Gerhard Wellein (2010). “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments”. In: *2010 39th International Conference on Parallel Processing Workshops*. IEEE, pp. 207–216.

Vef, Marc-André et al. (2020). “GekkoFS—A temporary burst buffer file system for HPC applications”. In: *Journal of Computer Science and Technology* 35.1, pp. 72–91.

Walsdorf, Oliver (n.d.). *Cisco UCS C240 M5 with Scality Ring*. `https://www.cisco.com/c/en/us/td/docs/unified_computing/ucs/UCS_CVDs/ucs_c240_m5_scalityring.html`. [Online; accessed 19-sep-2020].

Weil, Sage A et al. (2006). “Ceph: A scalable, high-performance distributed file system”. In: *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 307–320.

Weinman, Joe (2011). “Mathematical proof of the inevitability of cloud computing”. In: *JoeWeinman. com*.

Welch, Brent and Geoffrey Noer (2013). “Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions”. In: *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, pp. 1–12.

Wikipedia contributors (2021). *Technological Convergence — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-September-2021]. URL: `https://en.wikipedia.org/wiki/Technological_convergence`.

Wittman, Markus (2014). *OSU Micro-Benchmarks*. `https://blogs.fau.de/wittmann/2014/09/osu-micro-benchmarks/`. [Online; accessed 19-July-2020].

Wu, Kan, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau (2019). “Towards an Unwritten Contract of Intel Optane SSD”. In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA*.

Yamato, Yoji (2016). “Cloud storage application area of HDD–SSD hybrid storage, distributed storage, and HDD storage”. In: *IEEJ Transactions on Electrical and Electronic Engineering* 11.5, pp. 674–675.

Yang, Seokwoo et al. (2019). “Performance improvement of apache storm using InfiniBand RDMA”. In: *The Journal of Supercomputing* 75.10, pp. 6804–6830.

Zadok, Erez et al. (2017). “{POSIX} is Dead! Long Live... errr... What Exactly?” In: *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

Zaharia, Matei et al. (2010). “Spark: Cluster computing with working sets.” In: *HotCloud* 10.10-10, p. 95.

Zhang, Yunqi et al. (2016). “Treadmill: Attributing the source of tail latency through precise load testing and statistical inference”. In: *ACM SIGARCH Computer Architecture News*. Vol. 44. 3. IEEE Press, pp. 456–468.

Zhang, Ziyu et al. (2021). “RDMA-based apache storm for high-performance stream data processing”. In: *International Journal of Parallel Programming* 49.5, pp. 671–684.

Zhao, Yong et al. (2014). “A service framework for scientific workflow management in the cloud”. In: *IEEE Transactions on Services Computing* 8.6, pp. 930–944.

Zheng, Qing et al. (2012). “Cosbench: A benchmark tool for cloud object storage services”. In: *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, pp. 998–999.

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Unterschrift:

Ort, Datum: