Bachelor Thesis

Analyzing I/O Efficiency in Zarr, HDF5, and NetCDF4

Lucas Philipp
lucas.philipp@st.ovgu.de

May 23, 2025

First Reviewer:
Prof. Dr. Michael Kuhn

Second Reviewer:
Dr. Jannek Squar

Abstract

Scientific data is often handled by highly specialized file formats, which all aim to solve the same fundamental problems; fast access times to data at the lowest latency achievable for some of the largest datasets currently in use. This is especially true for earth data analysis, where petabytes of data are processed daily, with three of the most prominent formats used being HDF5, NetCDF4 and more recently Zarr. All of them have their own respective benefits and shortcomings, offering a variety of features, some shared between, some unique to them. Inherently knowing which format to use best for certain use-cases is not straight forward. They need to be evaluated carefully to carve out the best suited contender, as the environments these file formats are used in heavily depend on the interactions between complex software and hardware stacks.

This thesis presents a benchmark framework to emulate different scenarios for HDF5, NetCDF4 and Zarr leveraging select features to potentially increase performance like chunking, subfiling, parallel I/O and asynchronous I/O wherever possible. Throughput measures provide a broader picture for areas where formats still have potential to grow or might be limited already by external factors such as file systems like Lustre. The benchmark is written in both Python and C to evaluate the influence different programming languages and concepts can have on performance, to evaluate the importance for code maintenance and future extensibility. As all formats inherently try to cover the same important aspects of scientific computing needs, these metrics will help guide recommendations for future developments.

The results showed that file formats in Python like NetCDF4 are superior to other formats and features for parallelized workloads, including their respective counterparts in C, while newer entries like Zarr can match or even outperform more longstanding formats. This supports the notion to switch away from more complex programming languages like C towards languages like Python.

Contents

Chapter 1.

Introduction

1.1. Motivation

As the amount of data needed to perform analysis grows ever more rapidly, so do the requirements on the infrastructure handling this data. The creation of digital file formats, that can ensure efficient analysis and access to scientific data, has been a part of the problem since the 1940s when computers first entered the digital age, with the Z3 by Konrad Zuse [Copeland, 2006]. Developing ever more performant formats, that fit the growing needs of modern computing and support future applications yet to come, while not being bound by current possibilities and limitations such as hardware, became a new frontier. Formats are required to be forward thinking in their architecture and features, as they move into and beyond systems generating petabyte-scale data. These vast amounts of data require fast compute, faster access and lower latencies. Storage costs on that scale are high and wasting time and resources could lead to increases in both monetary and environmental factors. It comes to no surprise that development of existing file formats for scientific data is still ongoing and even new formats are being developed in areas where older, established formats simply do not fulfill the needs. Three of the most commonly used formats in scientific applications are HDF5, NetCDF4, and more recently, Zarr as the scientific community moves from C to the Python programming language due to its lower complexity.

As these formats have settled into the toolkits of developers as well as researchers and have become standard practice for certain applications; newer features and developments within their respective specifications might stay largely unexplored as evaluation can be complex and time consuming. There is a limited amount of knowledge regarding the potential interchangeability of different file formats and their respective performance characteristics especially across programming languages, which might additionally be affected by biases. Thus older, more established file formats stay in use even if they themselves could offer improvements with newer features or might even be outmatched by newer, alternative formats. If modernization is currently needed at all, in the form of swapping formats, or if processes can simply be extended by newer features with relatively little effort, is unclear.

1.2. Summary

This work aims to provide a simplistic implementation of a benchmark framework to compare currently existing file formats against one another leveraging their basic as well as extended configuration options and features. These include contiguous and chunked configuration,

comparison of implementations in different programming languages and different features provided within each format. The aim is to provide a deeper understanding of the advantages and disadvantages HDF5, NetCDF4 and Zarr offer and which situations might require different configurations to better take advantage of their respective features.

This will be explored starting with the history and developments all of them have undergone throughout the years in Chapter 2, followed by a comparison to existing attempts to cross-examine performance in Chapter 3. A different, broader benchmark framework will be introduced in Chapter 4, which will outline an approach to perform somewhat standardized benchmarks on cluster systems running a *slurm* environment such as *Levante* owned and operated by the "Deutsches Klimarechenzentrum" (DKRZ). Levante represents one of Germany's most powerful supercomputer systems and therefore provides an ideal opportunity for such analysis. Chapter 5 will evaluate different configurations of the benchmark with Chapter 6 detailing the conclusions that can be drawn thereafter. Associations between results will be offered in order to provide a deeper understanding for current processes and how they can be optimized. For example what these benchmarks mean for file-format-specific implementations of features, that can vary by programming languages such as *C* or *Python* and how feature completeness across them might affect future developments.

Performance characteristics of file formats in regards to input/output (I/O) especially the requirements on networking, compute and storage in cluster systems will be shown using standard metrics and tools.

Chapter 2.

Background

In this chapter an extensive background will be given. Important aspects covering the environment, this thesis will be focussing on, are going to be covered first, followed by an overview of file formats as a whole. This extends into more specific formats such as NetCDF4, HDF5 and Zarr. During the later portion of this chapter, common features will be explored to cover the similarities and miniscule differences of each format.

2.1. High Performance Computing

HPC as a technology represents a myriad of processors laid out in clusters working in parallel [Susnjara and Smalley, 2024]. Such a system is used to perform analysis processing massive, multidimensional datasets at high speeds. As they are purpose-built for these specific tasks and highly specialized, all components comprising such a system need to be carefully selected to allow for optimal performance. Clusters are usually composed of thousands of nodes, which represent high speed compute servers and centralized schedulers that manage the parallel computing workloads. Nodes are made up of one or more *central processing units* (CPUs) and optionally one or more *graphics processing units* (GPUs).

2.2. Input / Output

These systems handle their external communication through some form of networking, more specifically InfiniBand in Levantes case [Mellanox-Technologies-Inc, 2024]. Input / Output (I/O), in that sense, encompasses the communication a system needs to perform in order to function as intended, e.g. between nodes and the schedulers mentioned in Section 2.1. However, I/O can also be as simple as using a keyboard as an input device and a monitor as an output device.

These I/O operations can be processed by two different kinds of mechanism, *blocking* and *non-blocking* [Buettner et al., 2009]. Blocking means a process will be waiting for an I/O call to finish before moving on, leaving CPU cycles unused, while non-blocking allows a process to continue without waiting, effectively allowing the previously unused cycles to potentially be used up by some other task like compute, while I/O continues in the background.

Both blocking and non-blocking operations can be performed *synchronously* or *asynchronously*. This means that blocking a process and waiting on the I/O call to finish, is happening at the same time for synchronous-blocking. An application is therefore blocked until a call

is completed, in the form of a data-transfer or error. During this state it does not consume CPU and simply waits for a response, making the implementation overall very efficient. Asynchronous-blocking is non-blocking I/O with blocking notifications, which is convenient for asynchronous notification of for example I/O-descriptors, but is not advisable for HPC due to relying on inefficient *select* calls [Jones, 2006].

Synchronous-non-blocking is a less efficient variant of synchronous-blocking, as the application usually has to perform a busy-wait by making numerous calls to await completion. Since it gathers data as it becomes available from the I/O device, synchronous-non-blocking is not waiting for all the data to be present before "finishing" a read. Therefore an error needs to be returned, if there is data missing, to indicate that the call has to be redone. Asynchronous-non-blocking works differently in that it indicates the read operation has been successfully "initiated", while it completes in the background. This time can now be used to perform other tasks or processing, like setup, post-processing or compute, before the data is actually needed [Jones, 2006].

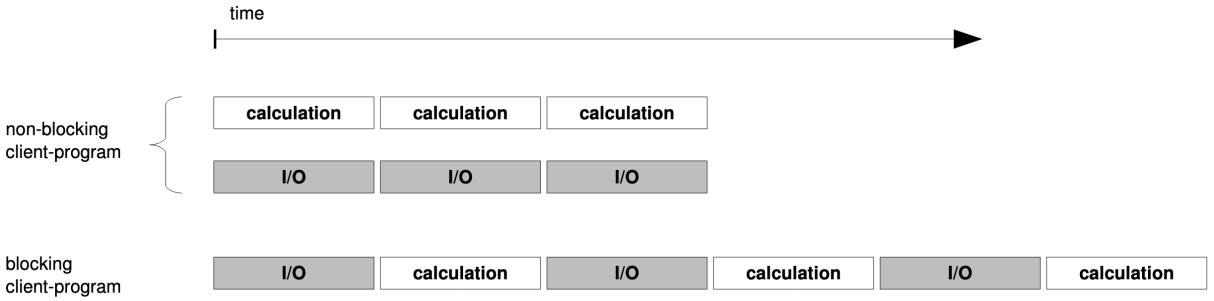
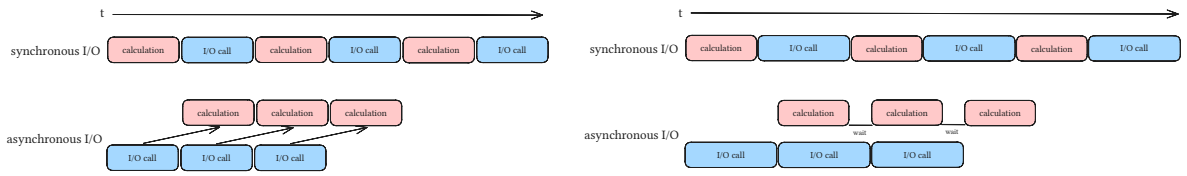


Figure 2.1.: Theoretical comparison of scenarios with equal times for I/O and calculation [Buettner et al., 2009]. Reprint with permission.

Generally the most used combinations of blocking / non-blocking and synchronous / asynchronous operations in a HPC context are synchronous-blocking and asynchronous-non-blocking. As such they will be referred to as synchronous I/O and asynchronous I/O respectively moving forward. Figure 2.1 presents what asynchronous (top) and synchronous (bottom) could look like in terms of making more effective used of resources by stacking operations. Both synchronous and asynchronous I/O can be done in parallel, allowing for multiple I/O operations to be performed at the same time.

However, Figure 2.1 does not illustrate how computation is handled if data, that I/O calls are being made to, is required for the calculation to continue. Simply what could be possible if calculation and I/O are unaffected by one another using asynchronous I/O, therefore exploiting the gap between slow I/O and fast calculations to handle both types of operations in a single process [Jones, 2006]. Considering a program trying to save the results of a calculation, this could be done while the calculations continue. This is unlike synchronous I/O, which would have to stop the calculation each time the program wants to save the results to disk, having to wait for the operations to finish, before restarting the calculation.



(a) Best Case Scenario

(b) With delay due to longer I/O call

Figure 2.2.: Synchronous I/O compared to Asynchronous I/O

If the data the I/O call is being made to, is needed for the calculation to finish, asynchronous I/O could still be beneficial. When loading a chunk of data a calculation is performed on, at some point a program needs to load the next chunk. With synchronous I/O this would need to interrupt the calculation or wait for it to finish, before being able to make that call. Asynchronous I/O however could start the call for the next chunk immediately after finishing the first call without the need to interrupt the calculation (Figure 2.2a). If all goes well the data could arrive before the current task finishes with the first chunk of data, effectively allowing the computation to continue uninterrupted. Even if the new data should arrive late (Figure 2.2b), the process of retrieving it has already started and is potentially almost finished, while it would have needed to wait of the calculation to finish first for synchronous I/O.

2.3. File Systems

File systems (fs) are an integral part of an operating system (OS), which handles file creation and access [Godoy et al., 2025]. They can be local or act as a distributed file system.

Server Classes	**Features**
Management Server	• Provide configuration information • File system registries
Metadata Server	• Record file system name spaces & inodes • Maintain file system index
Object Storage Servers	• Record file content in distributed binary contexts

Table 2.1.: Lustre Servers

One such distributed file system, most commonly found in HPC environments, is Lustre[1]. Lustre offers scalability across thousands of compute nodes and advanced I/O mechanisms to provide a high performance file system capable of handling modern computing needs. It is entirely implemented within the Linux kernel and relies on a client-server network architecture, designed to allow multiple clients to be managed by an appropriate, yet smaller number of servers. These servers are connected via network interfaces, which provide communication between them. Lustre offers different kinds of servers (Table 2.1) for different, more specialized tasks [OpenSFS and EOFS, 2017].

[1] `https://www.lustre.org/`

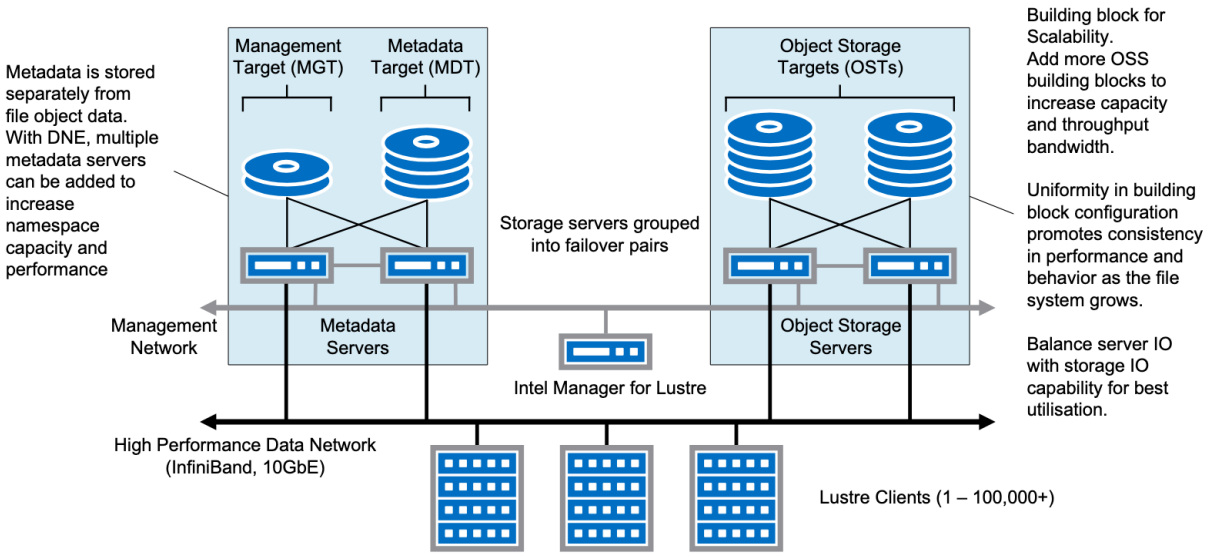


Figure 2.3.: Lustre Building Blocks

It separates metadata storage (inode) from block data storage (file content) (Figure 2.3) where all metadata operations such as creating or deleting a file, managing permissions, etc. are handled and stored by metadata servers, so-called Metadata Targets (MDTs), whereas Management Servers store Lustre configuration information on Management Targets (MGTs) [OpenSFS and EOFS, 2017]. Object servers handle writes of data content to persistent storage and can be written to concurrently, while individual files can be distributed across multiple objects on multiple servers. These are referred to as Object Storage Targets (OSTs). All forms of I/O communication between Lustre components passes through Lustre networking (LNet).

Additionally, Lustre can be configured with *Progressive File Layouts* (PFL), which is intended to simplify Lustre's I/O mechanisms and therefore the configuration of an optimal layout of data for a file. Users can easily achieve reasonable performance without the need to fully grasp Lustre's usage details [OpenSFS and EOFS, 2025]. PFL is characterized by increasing the stripe count of a file in a stepwise manner as the offset increases. Stripe count hereby refers to the number of OSTs a given file is striped over. The larger the individual file, the more OSTs are being used to store the file, which can help achieve better performance for both concurrent shared-single-file I/O and parallel I/O. Striping hereby refers to the process of segmenting a file into stripes in order to store these segments on different physical targets such as OSTs.

2.4. File formats

File formats, sometimes also referred to as data formats, represent a somewhat standardized way of describing how data is laid out and referred to within a file, in storage and / or memory. This is done by the process of encoding. A format handles the creation, storage and retrieval of data in a dependable and safe way, ensuring low latency, fast access times and availability of data. Usually file formats offer a specification that outlines functionality provided by the format such as encoding / decoding capabilities, support for compression or specialized layouts and a general definition of metadata required, such that, when followed, the format can be implemented and used on a wide variety of systems.

Not every type of file format has to support all kinds of data or their respective representations that may exist. Formats that can handle different kinds of data usually act as containers for other file formats such as Matroska (MKV) [Adobe, 2025] or allow conversion between their own specification and the target file formats specification to ensure interoperability.

Today there are many use-cases that not only require file formats that can handle different kinds of data or their representations, but additionally need to be optimized for specific tasks, as more generalized formats do not offer the performance required. One such example is for earth system data analysis. Thus more specialized implementations were created to allow for better handling of specific data, while also maximizing system performance. Some of those formats more prevalent in earth system data analysis are HDF5, NetCDF4 and more recently Zarr.

As the DKRZ mostly relies on these three formats, they have been chosen for further evaluation.

2.5. NetCDF

Network Common Data Form (NetCDF), released in 1990, is a data abstraction for storing and retrieving multidimensional data developed by Unidata. The goal was to create a machine-independent interface for capturing, managing, analyzing and displaying weather data [Rew and Davis, 1990]. Additionally, support was added for random access, to increase efficiency when accessing small portions of large data files. Wanting to closely resemble NASA's Common Data Format (CDF), Unidata considered extending CDF with an additional C interface and UNIX implementation in the 1990s. They learned that CDF had already been independently extended with a C interface and ultimately chose to refine said extension in the following NetCDF abstraction, also adding and adapting Candis C package, an analysis and display software for scientific data using UNIX pipes-and-filters approach.

NetCDF models scientific data as a collection of named multidimensional variables along with their coordinate systems [Rew and Davis, 1990]. Variables are specified by a list of named dimensions and a set of properties, each containing a type and shape, supporting most common types such as arrays, scalars, characters, integers and floating-point numbers. NetCDF also supports unlimited dimensions.

2.5.1. NetCDF Specification

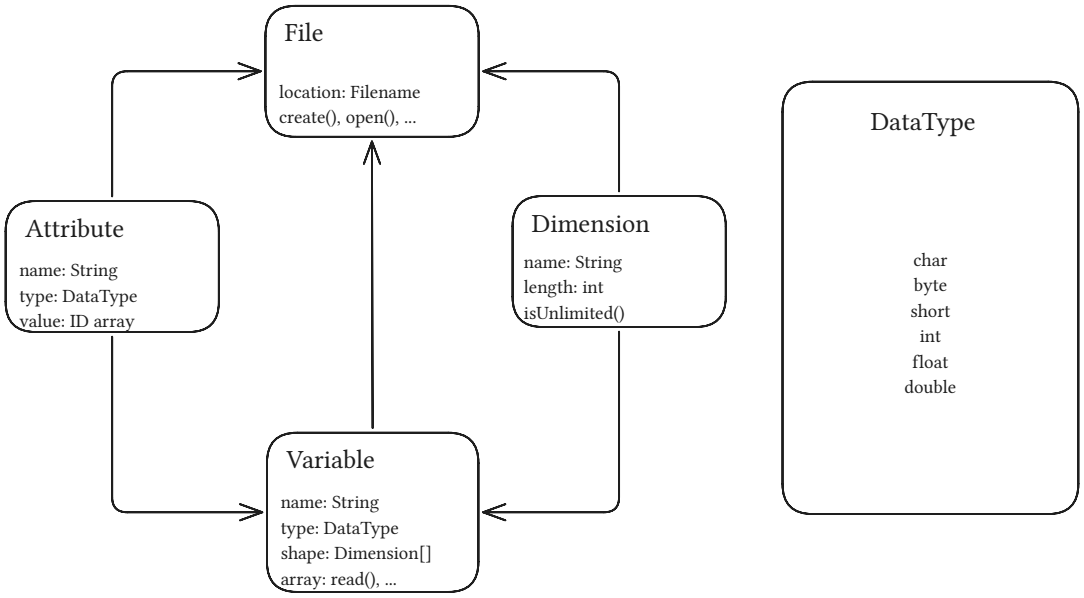


Figure 2.4.: NetCDF abstract file structure of the *Classic Data Model*. Partially redrawn [Rew et al., 2025a].

Since 1990, the NetCDF abstraction has undergone various version and feature changes, while continuing to support backwards compatibility for accessing files in previous iterations [Unidata, 2018a]. The initial format created in 1989 and released in 1990 as seen in Figure 2.4 is referred to as CDF-1. This *Classic Data Model*, as Unidata describes it, is still in use today, even after it has been extended in NetCDF version 4. It offers unlimited dimensions with a selection of 6 primitive data types including char, byte, short, int, float and double [Rew et al., 2025a]. In 2004, CDF-2 was released adding support for 64-bit offsets, which allowed for larger datasets to be created and used.

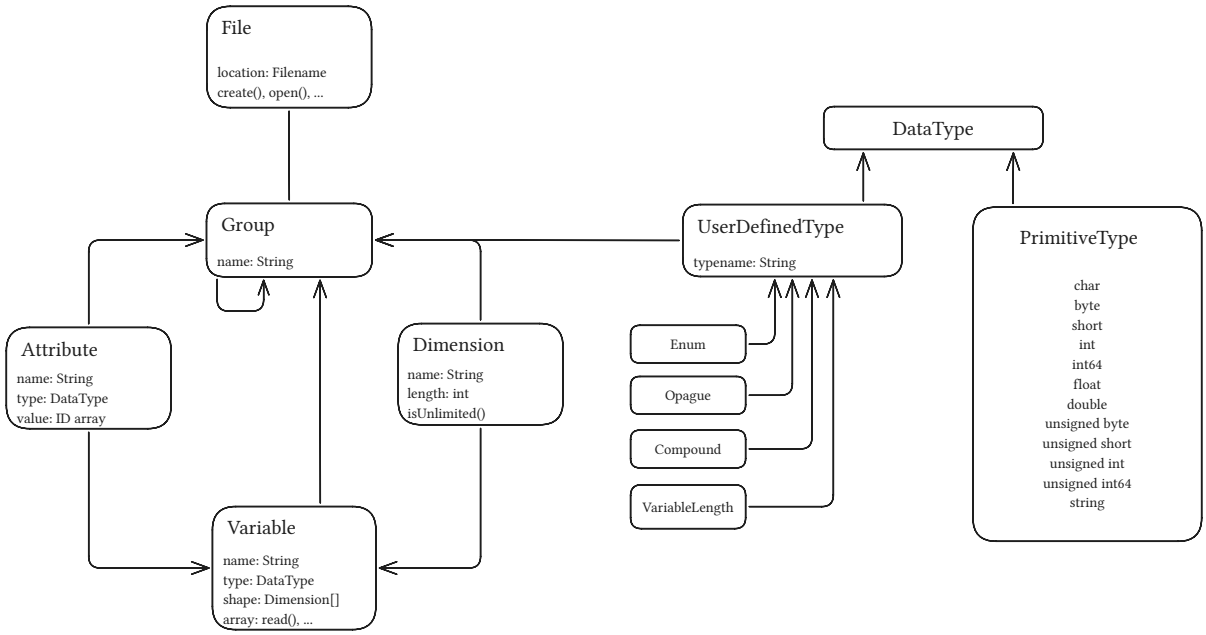


Figure 2.5.: NetCDF4 abstract file structure of the *Enhanced Data Model*. Unsigned data types were added in 2016. Partially redrawn. [Rew et al., 2025a]

NetCDF-4 was released in 2008 and referred to as such. It switched to using HDF5 at its core by layering an enhanced access interface with an *Enhanced Data Model* [Rew et al., 2025a] (Figure 2.5) to increase performance, adding per-variable compression, multiple unlimited dimensions and more complex data types. The new data model, also now known as the *Common Data Model*, was part of an effort at Unidata to find a common ground in scientific data solutions. Therefore groups and *user-defined types* were added, which allowed users to have more control over the type of data they wanted or needed to work with, while retaining Attributes, Variables and Dimensions from the *Classic Data Model*. The introduction of groups also added the ability to organize data in a hierarchical fashion.

Since Unidata was aware, that not all users rely on the added feature-set, they also release NetCDF-4-classic data format (Figure 2.4) providing the same performance benefit but stripping the added complexity for the sake of simplicity [Rew et al., 2025a]. In 2016, 64-bit data format was introduced with CDF-5, adding support for large variables and new data types like unsigned.

Throughout the years, those changes also brought some confusion surrounding NetCDF and its features. To add to the somewhat inconsistent naming conventions, some users and even Unidata itself, refer to NetCDF classic CDF-1, 64-bit offset CDF-2 or 64-bit data CDF-5 format as NetCDF-3-format, while referring to either the NetCDF-4 or NetCDF-4 classic model format as NetCDF-4 [Unidata, 2018b].

Additionally, while HDF5 is supported and used as the underlying core of NetCDF since 2008 and NetCDF-4 files can be converted into HDF5 files, NetCDF-4 does not support the full range of HDF5 capabilities, choosing to only support an initial implementation of a simpler HDF5 model. This means that some HDF5 files cannot be converted to NetCDF-4 and not every NetCDF-4 file can be expected to support the full range of HDF5s features using HDF5

tools. A list of not supported features can be found in the NetCDF documentation [Rew et al., 2025b].

2.6. HDF

The Hierarchical Data Format (HDF) is a self-describing means of sharing scientific data. It is designed to be easily shareable between different systems, projects and people [The HDF Group and Illinois, 2017]. Designed by the National Center of Supercomputing Applications (NCSA), the first version of HDF was implemented in spring-summer of 1988 and some months later an interface for scientific data was added supporting multidimensional arrays and related data. HDF was designed to be a format able to satisfy the needs of the NCSA, as no alternatives existed within the public domain at that time, which were suitable for scientific data and offered extensibility.

2.6.1. HDF Specification

Due to various changes in and around the HDF Group, most historical documentation on the development of HDF has seemingly been lost in recent years, as broken links, missing files or pages and missing documents riddle the Internet and the HDF Group's own website. This section will try to reconstruct its history from 2001 and onward as best as possible.

The first version of HDF, released in 1988, included a general-purpose interface and an 8-bit raster image interface. In the same year an interface for scientific datasets (SD) was added for multidimensional arrays and related data followed by the ability to store color palettes, 24-bit raster images and annotations [The HDF Group and Illinois, 2017].

In 1989, support for a general grouping structure and unstructured data was added leading to Vsets, which was implemented as a separate HDF library. That same year it became clear that the existing implementation was not sufficiently powerful thus sparking a need for a full overhaul of existing code. The first version to make use of these changes was released in 1992 as HDF Version 3.2. This was quickly followed by Version 3.3 in 1993 providing a new SC interface allowing for multi-file access and unlimited dimensions for arrays, alternative physical storage methods through tags, JPEG data compression, changes to Vsets and access to NetCDF files.

HDF4 was then released in 1996 allowing for n-bit integers and SDS compression, limited support for reading CDF files, a parallel I/O interface for CM5, auto configuration and more. Version 4.1 quickly followed in 1997 introducing data chunking, with chunking support being finalized in the year 2000.

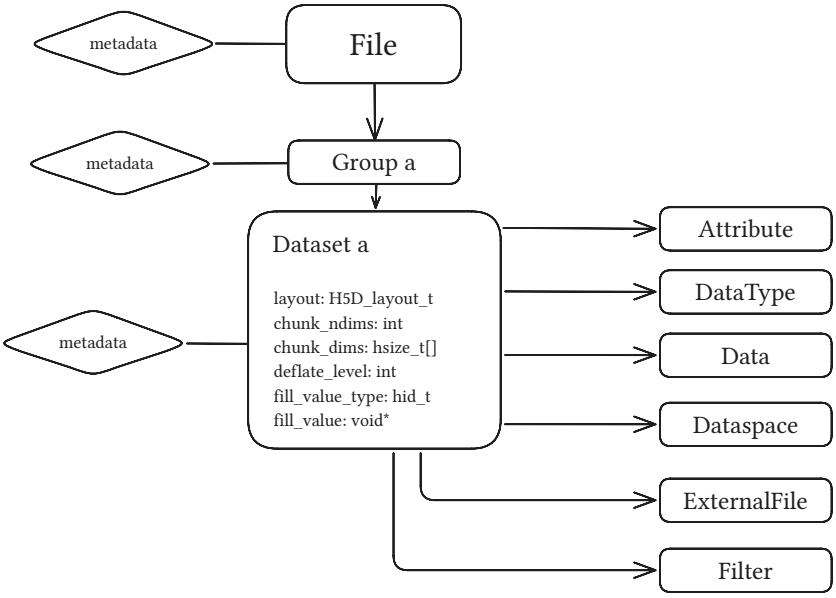


Figure 2.6.: HDF5 abstract file structure. Combined and redrawn [HDFGroup, 2025a].

Around the same time development on HDF5 began, which had a drastically different structure than HDF4, as there was a clear need to support files that exceeded 2 gigabytes. While HDF4 had support for basic objects such as scientific datasets (SD), 8- and 24-bit raster images, annotations, etc., HDF5 had only two primary objects: a dataset and a group. This made HDF5 much simpler while still being able to conceptually map HDF4 data to HDF5. It was initially released in 1998 as a serial implementation adding a parallel implementation in the following year. Figure 2.6 depicts a generalized HDF *Abstract Data Model* [Folk et al., 2011; HDFGroup, 2009; 2023].

Both HDF4 and HDF5 are actively being maintained by the HDF Group.[2]

2.7. Zarr

Zarr is a relatively new data format released in 2016. It is a community driven project which aims to create a specification and software for storing large N-dimensional, typed arrays known as tensors. Zarr focuses on distributed systems like cloud object storage, which can be accessed in an efficient and parallel manner. Data is represented hierarchically in a key-value-pair similar to HDF5 [Zarr, 2022]. This is referred to as a *Store* by Zarr, which encompasses all metadata documents and encoded chunk data as raw bytes. Such a store can be represented as a directory in a file system, with keys being the file-names and values being the file-contents which themselves can be read, written to or deleted via the operating system [Miles et al., 2025a]. Since HDF5 did not support parallel I/O with compression and thread-based parallelism at the time, Zarr was created to support compressible parallel I/O and compute on tensor storage [Miles, 2019].

[2] `https://www.hdfgroup.org/`

2.7.1. Zarr Specification

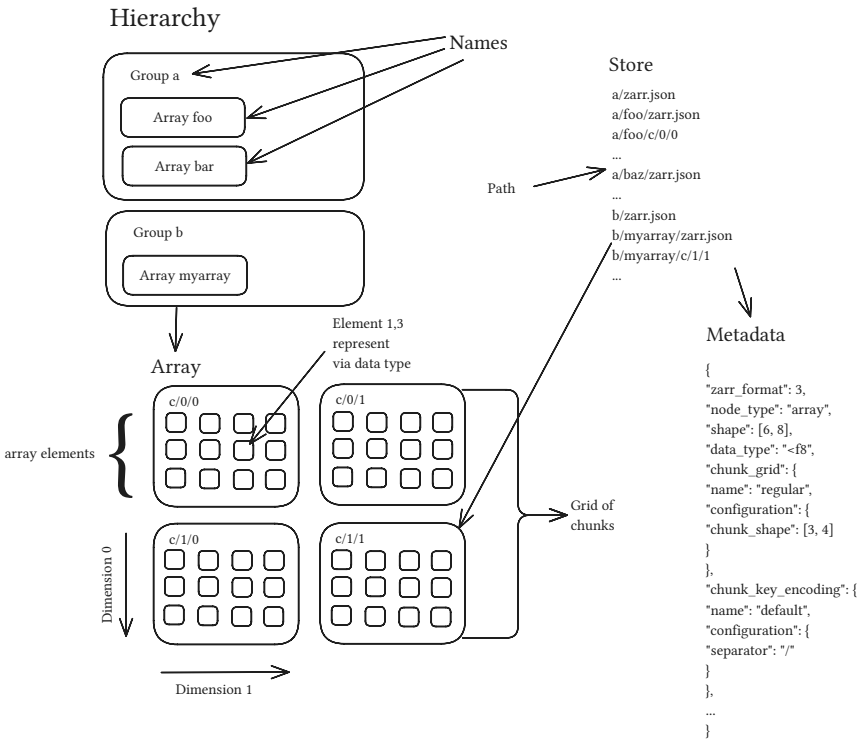


Figure 2.7.: Zarr abstract store structure. Redrawn. [Miles et al., 2025b]

In Figure 2.7, a Zarr store structure is illustrated: each Zarr *Store* stores essential configuration metadata, similar to NetCDF and HDF, as a JSON object. This ensures that data can be interpreted correctly on any system implementing the Zarr interface. This interface has undergone some changes from version 1 to the current version 3. Initially released as v1 in 2016 with support for chunking, compression and support for N-dimensional arrays, version 2 followed the same year just a couple months later extending Zarr's capabilities with a way to represent hierarchical storage using groups and filter support.

Version 3 specification was accepted in 2023 and has since been released in 2025. Notable changes from version 2 include explicit support for extensions and thus choosing to natively support a smaller number of data types as v2 by default, which can be added back through extensions [Miles et al., 2024; Zarr, 2022; 2024].

2.8. Common features between NetCDF, HDF and Zarr

	NetCDF	HDF	Zarr
Current Major Version	v4	v5	v3
Native Implementation	C, C++, Fortran	C, C++, Fortran	Python, C[3]
Open Source	✓	✓	✓
Self-Describing	✓	✓	✓
Hierarchical	✓	✓	✓
N-dimensional	✓	since HDF v3.3	✓
Compression	✓	✓	✓
Parallel I/O	since NetCDF-4	✓	✓
with Compression	since HDF5 v1.10.2	since HDF5 v1.10.2	✓
with MPI	✓	✓	×
Chunking	since NetCDF-4	since HDF4 v4.1	✓
Asynchronous I/O	×	with hdf5-vol-async	✓
Subfiling	×	since HDF5 v1.13.2	✓

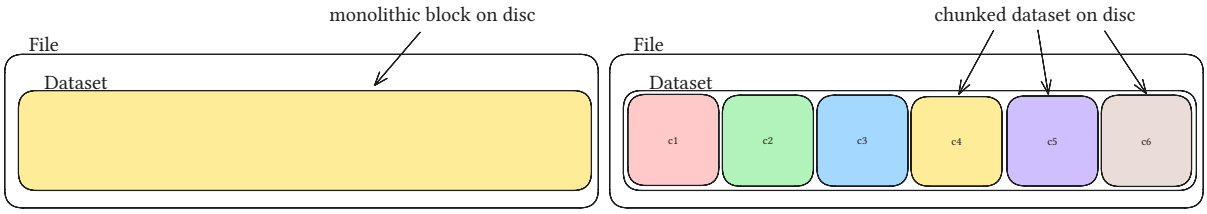
Table 2.2.: Comparison of functionally between Zarr, HDF5 and NetCDF4.

As can be seen from Table 2.2, all three formats chosen are quite similar in what they are trying to offer and achieve. Yet there are some points where they differ slightly, which might prove to be a disadvantage for certain use-cases. This is especially important when availability of a certain feature heavily depends on versioning and platform as for example with asynchronous I/O, which for HDF5 is only available through an external *virtual object layer* (VOL) [HDFGroup, 2025b] in the C programming language.

To better understand the differences and similarities, some features will be looked at in greater detail during the following sections, beginning with chunking.

[3] Zarr was natively implemented within NetCDF as NCZarr

2.8.1. Chunking



(a) Contiguous layout

(b) Chunked layout. Does not represent the actual layout of chunks on disk.

Figure 2.8.: Contiguous vs Chunked layout of data within a file

Chunking refers to datasets and the data within a file being divided into (usually) equally spaced chunks (Figure 2.8b) to increase performance. Working on smaller chunks of data is much more performant for computation, since only a subset of the data needs to be accessed instead of the whole contiguous layout. A contiguous layout refers to datasets in a file being serialized in a monolithic block on disk as seen in Figure 2.8a [HDFGroup, 2025c].

Chunked files can also be distributed better among the file system as a whole, which can allow file systems like Lustre to take full advantage of their architecture. This is called Alignment as chunking can be used to create smaller, more evenly spaced "blobs" of data within a file, which can be mapped to stripe-sizes used by the file system.

2.8.2. Parallel I/O

All three formats claim to support some form of parallel I/O, but it should be noted that parallel I/O and parallelism as a whole can be facilitated through multiple different means. Most notable for these formats are thread/process-based parallelism and the *Message Passing Interface* (MPI) [Regents, 2023]. While Zarr currently only supports and recommends thread/process-based parallelism through Python's native functionality, both HDF5 and NetCDF4 natively support parallelism through MPI. Any common implementation of the MPI standard, such as MPICH[4] or OpenMPI[5], can be used with both. However this does not mean Zarr can not be integrated with MPI within an application, it simply means Zarr does not offer a native, first party interface for file handling via MPI like HDF5 or NetCDF4 do and therefore requires a little more care, than setting a few flags, when trying to incorporate MPI with Zarr.

2.8.3. Asynchronous I/O

Asynchronous I/O (async I/O), as described in Section 2.2, is only available in HDF5 C through the use of the hdf5-vol-async VOL [The Regents of the University of California, 2020] and Zarr through native Python functionality. While NetCDF uses HDF5 at its core since the NetCDF4 major release, it does not support the full range of HDF5 functionality as mentioned in Section 2.5.1. This includes the Virtual Object Layer (VOL), which is an abstraction layer

[4] https://www.mpich.org/
[5] https://www.open-mpi.org/

that allows users to intercept API calls in HDF5 and implement storage that better suits the current applications data needs [HDFGroup, 2025b].

2.8.4. Subfiling

Subfiling is the concept of splitting a single file into multiple smaller files of equal size, which aims to find the middle ground between a single shared file and a file-per-process approach [Henderson and Breitenfeld, 2022]. There are two formats featured that support subfiling, though both use somewhat different representations of what a subfile encompasses.

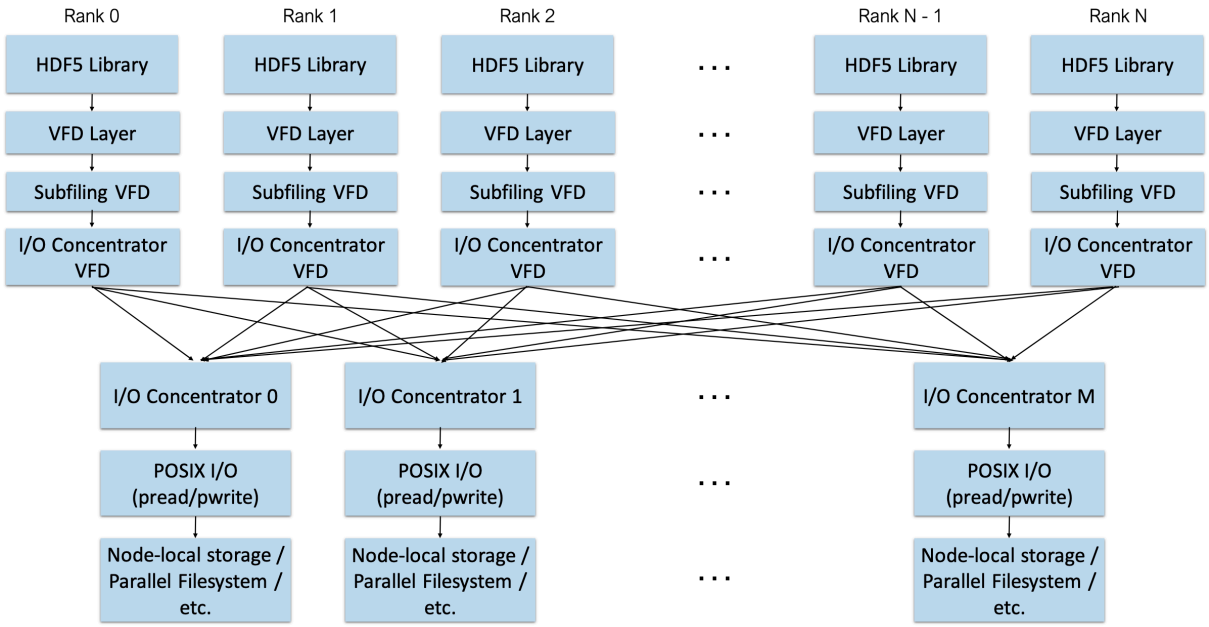


Figure 2.9.: Subfiling architecture [Henderson and Breitenfeld, 2022].

HDF5 for instance, supports Subfiling through a *virtual file driver* (VFD). This allows users to define a mapping between the HDF5 address space and the underlying storage altering the library's behavior [Henderson and Breitenfeld, 2021]. As an MPI-based file driver, it uses a subset of total MPI ranks as *I/O concentrators*. An MPI rank describes a unique process [Regents, 2023]. They are used to control the subfiles and manage I/O worker thread pools. Non concentrators forward I/O to the concentrators based on a logical offset in the HDF5 file. This means if a rank making an I/O call requests data, the driver will find and assign the I/O call to the I/O concentrator responsible for the specific subfile the data is located in as illustrated in Figure 2.9. By default one subfile per machine node is created using the Subfiling VFD.

Zarr, on the other hand, creates subfiles equal to the amount of chunks that are created for a given file per dataset, as these represent the key-value-pair in a Zarr store (Section 2.7.1). This means each of these subfiles contains a single chunk of the full dataset.

Chapter 3.

Related Work

In this chapter, related work will be covered, that tries to achieve or perform similar comparisons. Their respective approaches will be covered in detail and results will be shown.

3.1. Python benchmark of Zarr, HDF5, NetCDF4

Similar benchmarks were performed by [Ambatipudi and Byna, 2023]. Measurements were taken across Zarr, NetCDF4 and HDF5 interfaces for *create / open* and *read / write* use-cases with tests being performed on contiguous datasets without utilizing chunking, asynchronous I/O or subfiling. Six different file sizes were tested with dataset counts ranging across 2048, 4096 and 8192 of either 128 or 256 elements per dimension of type *float32* equating to total file sizes of 1MB, 4MB, 8MB, 128MB and 16GB tested.

The benchmark consists of two components. One part compares the time taken to create a dataset and write data to a dataset and a second portion, which opens the dataset at a later time and reads its contents. This can be configured using a supplied *.yaml* file, including parameters such as number of datasets to create within a given file and dimensions of the array written to each dataset. Measurements are then stored in a *.csv* for later evaluation.

To mitigate caching effects the script places files into a named directory. It is then copied to another directory and renamed, once the file is fully created. The *write* portion of the benchmark measures time taken for each dataset to be created and populated with random *float32* elements and then divides that time by the number of datasets to find the average. In reverse the *read* portion measures the time taken to open all datasets and then the time taken to print their contents to *standard output* (*stdout*), ending again with dividing by the number of dataset averaging the time taken for opening and reading one dataset.

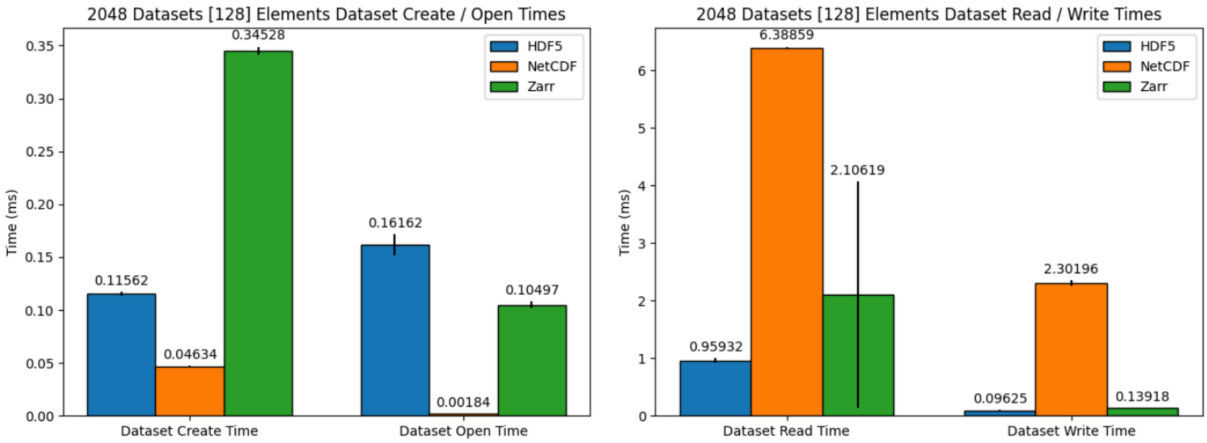


Figure 3.1.: Results for Dataset Create, Open, Read and Write [Ambatipudi and Byna, 2023]

Example results can be seen in Figure 3.1 split into the aforementioned *Create / Open* and *Read / Write* portions.

Findings concluded in HDF5 being superior for read / write, while NetCDF4 is generally better for creating and opening a dataset with Zarr usually trailing the performance of HDF5, demonstrating a method to compare format using the Python API.

3.2. HDF5 subfiling presentation

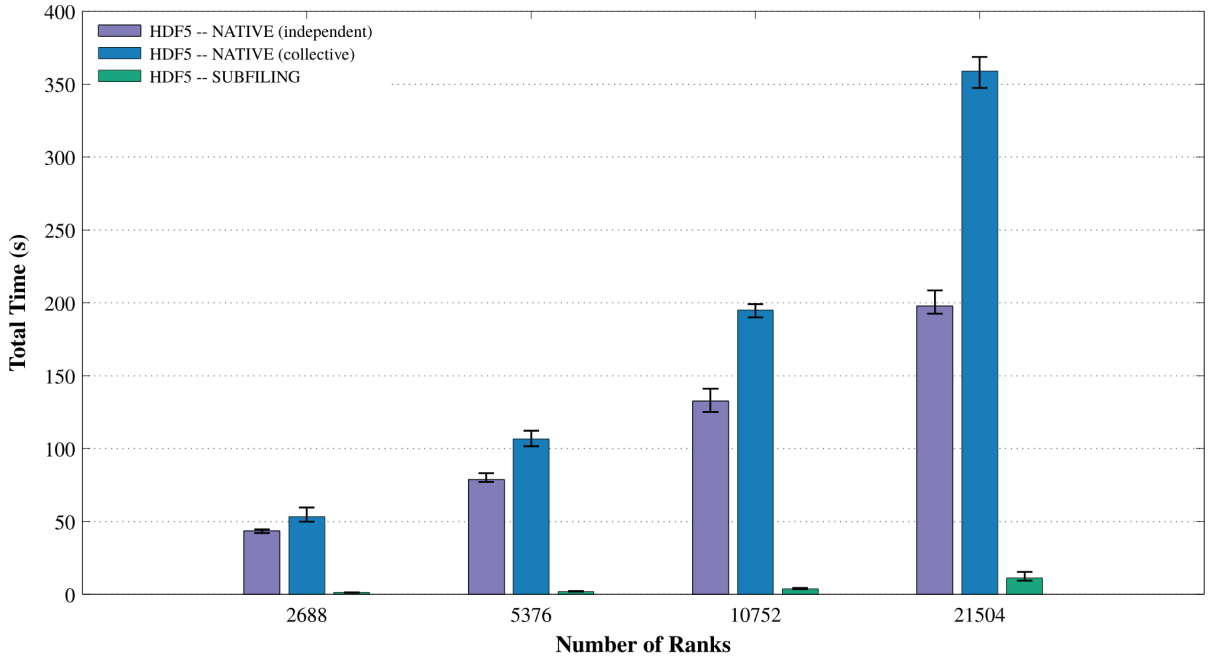


Figure 3.2.: CGNS Benchmark_hdf5, Summit (Four Runs Per Process Size) [Henderson and Breitenfeld, 2022]

When the HDF5 group introduced subfiling in 2022, they also performed some benchmarks against regular parallelized HDF5 with MPI [Henderson and Breitenfeld, 2022]. They compared against both independent I/O and collective I/O and found greatly improved performance for 2688 ranks up to and potentially beyond 21504 ranks (Figure 3.2). The benchmark

used writes and reads mesh coordinates, elements connectivity and solution data. A mesh of 130 million is used for the 21,000 ranks as 6-node pentahedral elements, which is halved as rank count decreases.

This results in a very small general filesize used considering the total number of ranks involved. For context on the DKRZs Levante, to use 21,000 ranks one would need to allocate about 164 nodes in total to simply read a 57 GB file.

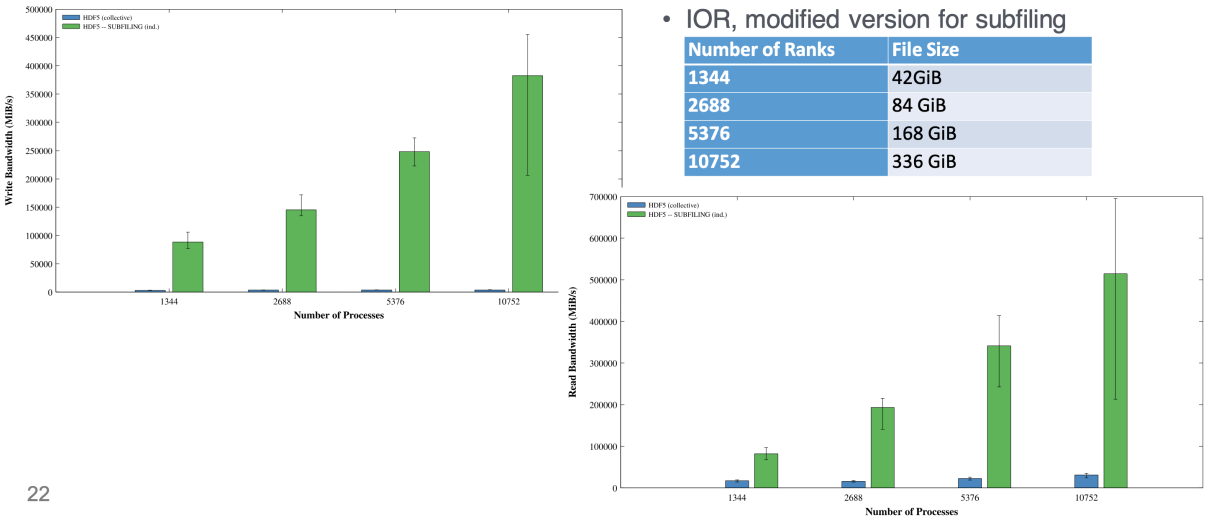


Figure 3.3.: IOR, modified version for subfiling [Henderson and Breitenfeld, 2022]

Nevertheless the results are quite impressive for bandwidth as well running IOR (Figure 3.3), with subfiling offering a staggering reduction in total time taken to read a file.

3.3. HDF5 and NetCDF4 benchmarks using IOR

[Bartz et al., 2015] performed performance analysis on NetCDF4 and HDF5 using Lustre to offer best practices when choosing I/O configurations in Lustre. They tested different access patterns, disjoint and interleaving I/O, aligned and unaligned access, independent and collective I/O with chunked and contiguous layouts.

A single IOR process was used per client node, as it was sufficient to saturate the network interface, to extract the maximum performance possible from the system. During reads / writes 20GiB were transferred per client node for a total of 200 GiB, exceeding the amount of memory available to prevent artificial caching. Transfer size is set to 1MiB to match the internal Lustre chunk size.

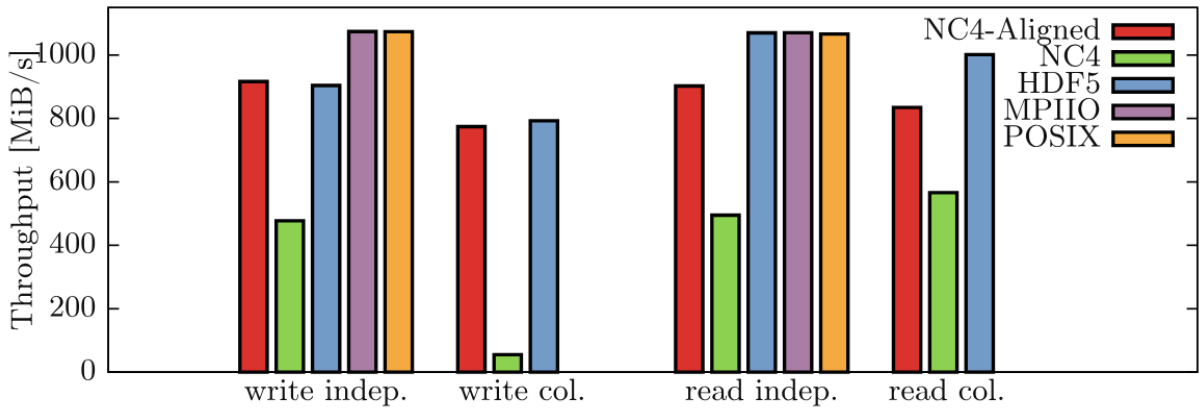


Figure 3.4.: 1-OST pattern [Bartz et al., 2015]

As an example (Figure 3.4) shows their results for a contiguous layout using 1 object storage target (OST). Both MPI-IO and POSIX are identical for independent read and write, while collective I/O suffers for writes due to unnecessary synchronization. NetCDF4 in general performs worse than HDF5 due to NetCDF4 not providing the ability to align access to Lustre's storage boundaries. It was shown that performance could be improved drastically, when resolving NetCDF4's shortcomings via a custom patch.

Chapter 4.

Implementation & Benchmarking

This chapter will cover the design and implementation of the benchmark framework, by first setting some baselines required for proper comparison. Afterwards the necessary steps for configuring and running the benchmark will be shown, detailing notable differences in code.

4.1. Baseline

Before benchmarking Zarr, NetCDF and HDF5, some things need to be considered, as the benchmark should ideally offer precise, carefully curated results later on. Therefore baselines need to be defined, which will help guide the design. These include important factors, like which tasks are being performed and measured, how these measurements are taken, what issues could potentially arise with the design and how they can be mitigated. For example caching effects an operating- or file-system might perform need to be cared for.

Beginning with the file formats used, which should support the programming languages chosen as well as all features leveraged. If features are used that are not supported by each format, they should only be compared to those formats that do support them. Comparisons across features are possible, but should only be done anecdotally. For example features like *subfiling* and *async I/O*, in the form of the *HDF5-VOL-Async* driver, only exist for HDF5 in the C programming language. HDF5 Python does not support them, nor does NetCDF4. Similarly Zarr lacks a native, first party C implementation and MPI support.

Additionally the benchmarks will be run and evaluated on the DKRZ cluster system *Levante*. *Levante* relies on *Lustre*, which represents one of the most commonly used file systems in HPC.

4.2. Benchmarks

Benchmarks are implemented in both Python and C programming language and perform *read-to-memory* tasks across a number of file sizes and configurations.

```
case "zarr":
  return self.dataset[variable][:]

case "hdf5":
  return self.dataset[variable][:]

case "netcdf4":
  return self.dataset.variables[variable][:]
```

Listing 4.1.: Python Syntax

For Python reading or writing a file is quite simple and fairly standard, as all formats adopted Numpy's *slicing* syntax (Listing 4.1) [NumPy Developers, 2025]. For HDF5 slices are translated to "hyperslab" selections.

```
// define id types for future reference
hid_t file_id, dset_id;

// allocated memory buffer
float *rbuf = calloc(some_size, sizeof(float));

// do file operations
file_id = H5Fopen("file-name", H5F_ACC_RDONLY, H5P_DEFAULT);
dset_id = H5Dopen2(file_id, "/X", H5P_DEFAULT);

H5Dread(dset_id, H5T_NATIVE_FLOAT, H5S_BLOCK, H5S_ALL, H5P_DEFAULT, rbuf);
H5Dclose(dset_id);
H5Fclose(file_id);
```

Listing 4.2.: C Syntax

C however requires at lot more care and thought to simply read a file (Listing 4.2). When looking at the HDF5 API, first id types need to be defined, which will be used to reference the file, dataset, group, etc. that has been created or is accessed later. Then a buffer needs to be allocated in memory, large enough to contain the data to be read. All of this information has to be passed to the HDF5 API. HDF5 splits its API into different sections, each handling a different area of the HDF5 stack.

This includes the file functions that perform operations on files like create or open and return a *file_id*, which are marked as H5**F**. Next the user needs to open a dataset using the H5**D** functions within a given file identified by its file_id. This returns a dataset id, which is used to identify the given dataset during the following read call. Here the type to read as *H5T_NATIVE_FLOAT* is provided along with the read buffer to read the data to. *H5S_BLOCK* is used as the *mem_space_id* to signal to the HDF5 API, that the buffer provided is a single contiguous block of memory with same number of elements, while *H5S_ALL* is used for the *file_space_id* to select the full file dataspace, as defined by the dataset's dimensions.

Finally the file needs to be properly closed, to avoid corruption or errors, when another process tries to open that file later on. NetCDF4 requires similar steps, however the exact API calls are different.

```python
# Python
# start timer
start = time.monotonic()

# perform read call
self.dataset[variable][:]

# end-timer and calculate elapsed time
results = time.monotonic() - start
```

```c
// C
// define timers
struct timespec start, end;

// initialize start-timer
clock_gettime(CLOCK_MONOTONIC, &start);

// perform require setup, open file, read call and close

// initialize end-timer
clock_gettime(CLOCK_MONOTONIC, &end);

// calculate elapsed time
double elapsed = end.tv_sec - start.tv_sec;
elapsed += (end.tv_nsec - start.tv_nsec) / 1,000,000,000.0;
```

Listing 4.3.: Time measurements taken for both Python and C

Time is measured, starting the timer directly before an abstract read call is being made and ending when the call finishes executing (Listing 4.3). This can include some setup, since some of these steps are inherently needed especially for the C APIs. Python's *time.monotonic()* and C's *clock_gettime(CLOCK_MONOTONIC)* API Call are used to get both reference points. This should ensure optimal time measurements being taken by providing a high resolution clock.

For Python different kinds of configurations can be requested. As an example one, two or three dimensional data can be requested for a given dataset. This is possible due to the simplified shaping Python offers, which all three file formats have adopted. The C portion of the benchmark currently only allows a single 1D dataset to be created with dynamically scalable file-sizes, due to the higher complexity of the language. These configurations can be freely adjusted in a *setup* section within *main_cluster.py*, which controls the benchmarks to run.

Main_cluster.py is designed for clusters using slurm. Since file-caching-mechanisms on such clusters can highly skew the benchmark results, as they allow an I/O call for a file (e.g. read) to be answered from faster cache or memory, if sufficient space is available. The benchmark should always read from disk storage to be representative of the actual performance a given format can offer.

For cluster configurations each benchmark and file creation is launched as a new subprocess, which authors a completely new node allocation on the cluster and waits for said allocation to free upon completion. Levante executes a clean-up script (Listing A.3) on each node, which is run after an allocation has been freed. This script frees all resources on that specific node, mitigating caching effects should said node be assigned to the benchmark script again.

```
setup = {
    "run01": {"X": (shape, chunks), "Y": ([1024], [64])},
}
```

Listing 4.4.: Setup example

Some form of setup is required to run the benchmark. A setup is defined as a Python *dictionary* of *runs*. Each *run* represents a file to be created by the benchmark, that describes which datasets will be created and of what shape. In Listing 4.4 an example setup is provided, which will create a file containing two datasets *X* and *Y*. These can be named arbitrarily only when running the Python part of the benchmark. Otherwise the dataset needs to be named X as the C portion is limited to a single dataset, which can be created within a file.

```
# Python
case "zarr":
  root.create_array(name=variable, shape=shape, chunks=chunks, dtype=dtype)
case "hdf5":
  root.create_dataset(variable, shape=shape, chunks=chunks, dtype=dtype)
case "netcdf4":
  root.createVariable(variable, dtype, dimensions, chunksizes=chunks)
```

```
// C
cdims[0] = chunk;
H5Pset_chunk(plist_id, 1, cdims);
```

Listing 4.5.: Chunking for Python and C

Each dataset requires its shape and chunksize to be known beforehand and will be Python *list* objects. The amount of elements specified in shape will then be created and will be of type *float64*. The file will be chunked if chunks are specified. For Python and C chunks can easily be set as shown in Listing 4.5. This is done to ensure that files, that are a multiple of *1GB*, can easily be created. Only the native Python formats can use multidimensional chunking. Additional context will be provided in Chapter 5. *Chunks* can be left empty to perform the default behavior defined by each format in that case. For Python *Zarr* this enables auto-

chunking as Zarr heavily relies on chunking for its native implementation, as explained in Section 2.8.4. Both Python *NetCDF4* and *HDF5* configure no chunking entirely, exactly like their C counterpart, as they utilize a contiguous layout by default. This ensures that all file formats behave exactly as intended when configured with the bare minimum parameters required.

```
/root
    /X: (shape, chunks), float64
    /Y: ([1024], [64]), float64
```

Listing 4.6.: Example structure of a file create by each benchmark

In Listing 4.6 a possible view of a file is represented, that could be create by the Python portion of the benchmark across all three formats.

```python
def runner_(df, variable, iterations):

    for i in range(iterations):

        # run benchmark on python file and wait to finish
        call = f"python benchmarks.py -b 1 -v {variable} -i {1}"
        p = subprocess.Popen(["sbatch", "slurm-scripts/run-anything.sh",
            call, "False"])
        p.wait()

        # read results from JSON and add results to Pandas DataFrame

    # delete python file

    return df
```

Listing 4.7.: Example runner script

Main_cluster.py offers multiple *runner_* functions, which encapsulates the logic for performing a specific benchmark and collecting the generated time measurements (Listing 4.7). These functions launch a subprocess that controls the slurm job and pass needed configurations to the benchmark scripts, which are then executed. There is another important function that is marked as *bench_variable*. As the name would imply *_variable* runs existing benchmarks for a given dataset named *variable* with its shape provided by *setup*. Benchmarks can be further configured by providing some number of *iterations* to run and *mpi_ranks* representing the number of ranks to use.

Each file format offers two modes of operation in the form of serial and parallel, which utilizes the *Message Parsing Interface* (MPI), except for Zarr. MPI was chosen as it represents a standardized approach to parallelize applications, that is widely used in HPC and is natively supported by NetCDF4 and HDF5.

```
  # only needed for NetCDF4
  self.dataset[variable}].set_collective(True)

  total_size = self.dataset[variable].shape[0]
  size = int(total_size / rsize)

  rstart = rank * size
  rend = rstart + size

  self.dataset[variable][rstart:rend:]
```

Listing 4.8.: Python MPI calls

A MPI call in Python for collective I/O using *mpi4py* looks similar to a serial one (Listing 4.8). The big differences lie with the range of elements that is read per rank, which is specified within a *slice* via *rstart* and *rend*. The total number of elements within a dataset is divided by the number of *Ranks* (*rsize*), ensuring each rank reads 1/Ranks of the current dataset. This allows for parallel to behave like serial and read back the entirety of the dataset. Additional mpi4py transparently handles the *MPI_Init_thread* and *MPI_Finalize* calls, so a user is not required to handle them manually.

```
// MPI_Init_thread and some more setup before
hsize_t process_mem_size = size / mpi_size;
count[0] = process_mem_size;
start[0] = count[0] * mpi_rank;

// following only needed for HDF5
memspace = H5Screate_simple(1, count, NULL);
filespace = H5Dget_space(dset_id);
H5Sselect_hyperslab(filespace, H5S_SELECT_SET, start, NULL, count, NULL);

// initialize buffer
float *rbuf = calloc(process_mem_size, sizeof(float));

// set property list to use collective I/O
plist_id = H5Pcreate(H5P_DATASET_XFER);
H5Pset_dxpl_mpio(plist_id, H5FD_MPIO_COLLECTIVE);

// perform read call
H5Dread(dset_id, H5T_NATIVE_FLOAT, memspace, filespace, plist_id, rbuf);

// close and MPI_Finalize
```

Listing 4.9.: HDF5 MPI calls

For C, the steps are mostly identical, except for HDF5 where some more HDF5 specific API calls need to be added (Listing 4.9). While NetCDF4 would only need to have some of its

API calls swapped to their *_par* equivalents, HDF5 requires a bit more setup. The important component here is the *hyperslab* as a portion of a dataset. A hyperslab can either be a logically contiguous collection of points within a dataspace or a regular pattern of points or blocks and requires four parameters [The HDF Group, 2006]. The first parameter is *start*, which describes the starting location, followed by *stride*, as the number of elements to separate each element or block to be selected, with *NULL* defaulting to a value of 1. *Count* describes the number of elements or blocks to select along each dimension and finally *block* specifies the size of the block selected from the dataspace.

Some MPI specific setup has to be done manually as well. *MPI_Init_thread* and *MPI_Finalize* both have to be called by the user and to set collective I/O, the *file access property list* of the HDF file needs to be set to *H5FD_MPIO_COLLECTIVE* (Listing 4.9). NetCDF4 requires similar changes during its `nc_var_par_access(ncid, varid, NC_COLLECTIVE)` call.

Zarr does not currently offer a native *MPI API* within its specification as of writing [Miles et al., 2025c]. Using MPI with Zarr should be possible and was considered, but has been omitted for this iteration of the benchmark due to time constraints.

Data is aggregated within a Pandas *DataFrame*, which will be used to interpret and plot the results using the Python *Plotly* library. A *JupyterNotebook* is provided for that purpose on Github[6].

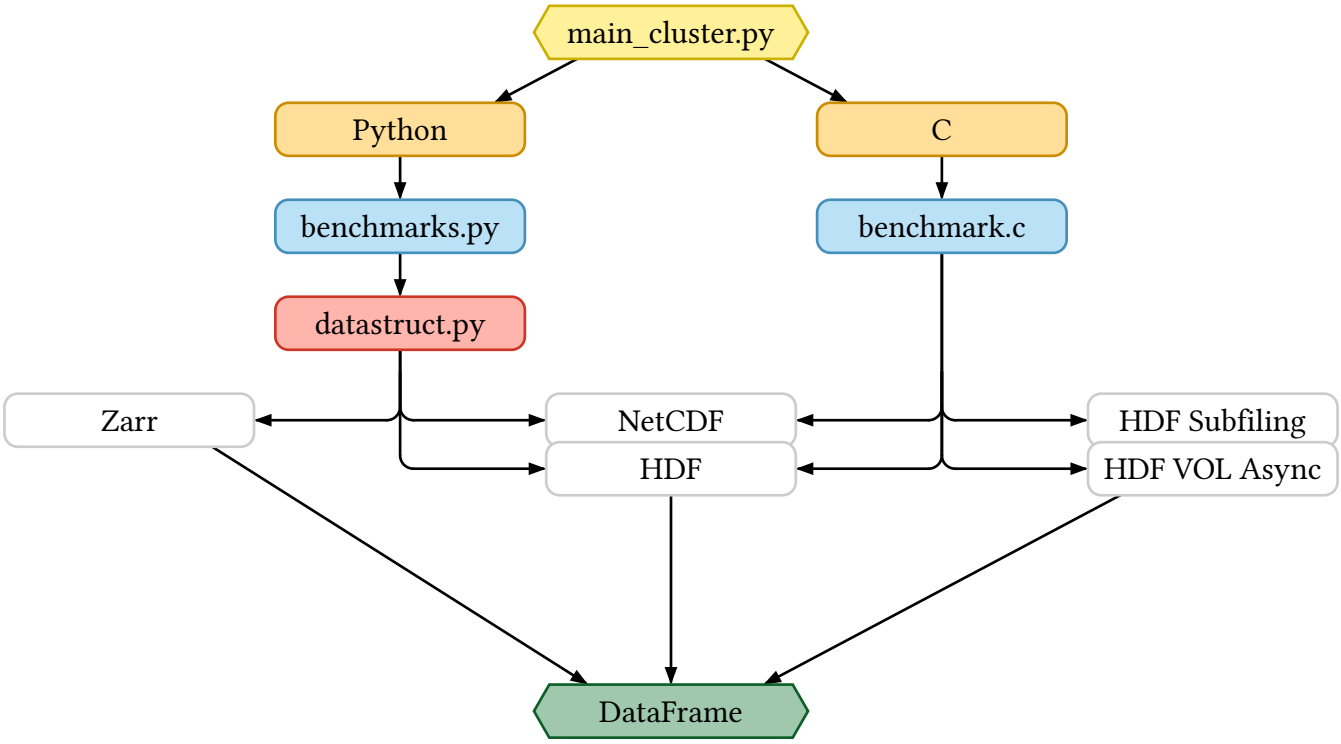


Figure 4.1.: Visualization of Benchmark flow

A depiction of steps, the benchmark takes, is shown in Figure 4.1. The logic for performing a call to one of the API's used, is implemented within *benchmark.py* and *benchmark.c* (), as the benchmark is designed to be extendable. All measurements are taken covering the respective

[6] `https://github.com/leuchthelp/dkrz_dev.git`

library calls (▭) for all file formats and features. They are then repeated a set amount of times in order to account for variance within execution times and nodes used. *Datastruct* (▬) represents a simple helper class to streamline the benchmark on the Python side.

Chapter 5.

Evaluation

To facilitate proper understanding of the environment the benchmark is run in, an extensive overview of the configuration used will be provided in this chapter. This is to ensure results can be reproduced by third parties. During the later portion, the results generated by the benchmark will be evaluated in detail. Followed by an extensive summary, which helps contextualize the results.

5.1. Setup & Benchmark configuration

In order to evaluate the results, some nuances should be considered. Each benchmark is run on Levante using the *compute* partition utilizing four nodes. Every node is configured with 2 AMD 7763 processors and 256 GB of RAM [© Deutsches Klimarechenzentrum GmbH, 2025]. Nodes are connected via a NVIDIA Mellanox InfiniBand HDR 100G/200G network interface with a theoretical maximum throughput of 12.5 GB/s.

Four nodes were chosen to allow HDF5 subfiling to function as intended by distributing its I/O Concentrators (IOCs) across selected nodes, creating one subfile per node. With one node selected only one subfile would be created, potentially harming performance.

In order to ensure all other formats are not negatively affected by increasing node count, reruns were performed with one node selected. Successive runs with one and then four nodes showed no real performance differences, staying within run-to-run variance. An example of what can be considered run-to-run variance will be given in Section 5.2. As such it was concluded that the other formats are not negatively impacted.

```
1  /root
2      /X: (shape, []), float64
3
```

Listing 5.1.: File created by default.

As mentioned in Section 4.2, each iteration of a single benchmark run, for a single feature, is launched as a separate node allocation using *slurm*'s *sbatch*. This allows for accurate measurements by mitigating caching effects. A sample sbatch script can be found in Listing A.2.

The file created, always contains a single, one dimensional dataset without chunking, unless noted otherwise (Listing 5.1). *Shape* represents the number of *float64* elements needed to create a file. For example when specifying *6,710,886,400* elements this will result in a *50 GB* file.

Extent	OST
0-1G	1
1-4G	4
4G+	16

Table 5.1.: PFL configuration.

Additionally Lustre is using Progressive File Layouts (PFL), which is configured as seen in Table 5.1.

Lustre offers a maximum single stream throughput of 1-3.2 GB/s, since it relies on Buffered I/O [Farrell, 2023], therefore all user data is copied through the page cache. This is a software limitation in the version of Lustre (2.14) currently being used.

For formats that feature the *-parallel* suffix, MPI based parallelism is leveraged and a standard configuration of 32 Ranks is the default unless noted otherwise. HDF5 *Async* and *Subfiling* rely on MPI, but are not specifically referred to as -parallel. Any form of MPI communication is set to *collective I/O*. Only exception being Subfiling, as it currently does not support collective I/O for non-metadata operations. For collective I/O multiple I/O calls can be combined into one larger operation and optimized by the MPI library. As subfiling was evaluated against collective I/O by the HDF5 Group (Section 3.2) and showed significant improvements, it was deemed agreeable to set collective I/O as the default.

Formats that lack the -parallel suffix are the *serial* default without any form of parallelism.

The *subfiling-driver* in HDF5 requires some configuration specific to this feature. A stripe-size of 32MB is set and aligned using *H5Pset_alignment* as per recommendation. Additionally 1 I/O Concentrator is configured per node with 4 worker threads per Concentrator as per the *HDF5-Subfiling-User-Guide* [henderson, 2023].

The software stack includes NetCDF4, HDF5 and Zarr packages for both the C and Python programming language (see Listing A.1) installed via *Spack*[7], an open-source package manager designed for HPC environments and Python's build in package manager *pip*.

Plots are generated using the Plotly Python library and can be found in *dev_original.ipynb*[6]. The time taken to read a single dataset at a given size is measured. This behavior is repeated five times and the arithmetic mean is calculated using Python Numpy *.mean()* [NumPy Developers, 2024]. Results are plotted as a line chart to provide an insight into how formats handle increasingly larger datasets. Additionally the mean-time taken is divided by the total size of a dataset as $\frac{\text{mean-time}}{\text{dataset size}}$ to evaluate throughput of the system using a given format.

5.2. Anomalies on Levante

Some anomalies have been observed as a result of inconsistent compute node behavior. Nodes are chosen from a single cell on the cluster as they become available. Therefore different nodes

[7] `https://spack.io/`

are used during different points of the benchmark. This has resulted in outliers which will be specifically mentioned. Some were observed on multiple reruns of the entire benchmark, showing at multiple file sizes with no clear pattern. Such it is concluded that some nodes within a cell can experience suboptimal performance, which affects only specific formats.

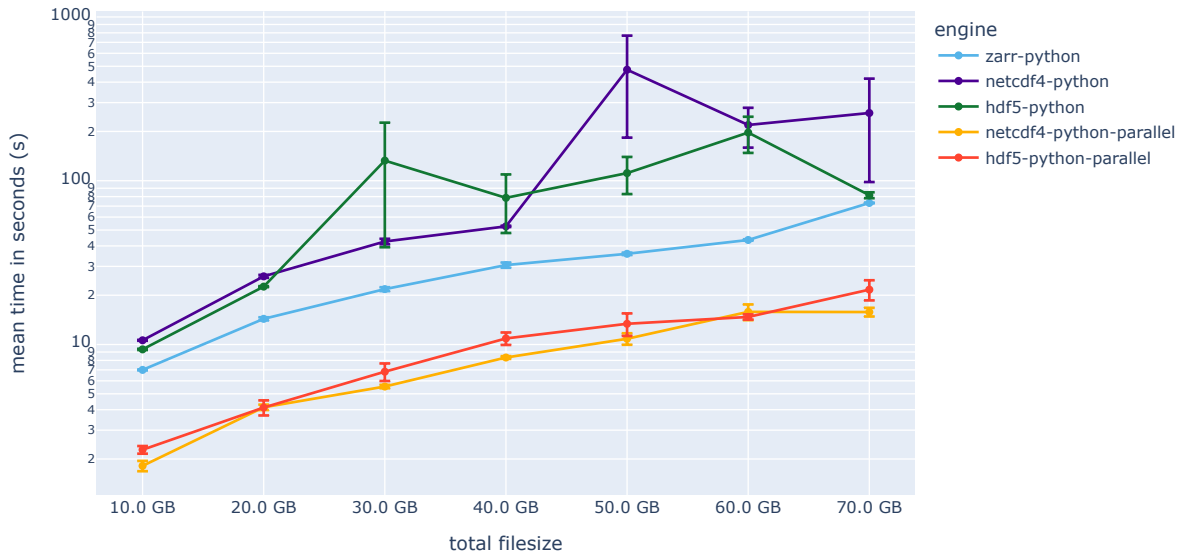


Figure 5.1.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

As an example in Figure 5.1 irregular spikes can be observed for both *hdf5-python* and *netcdf4-python*. No other formats seem to be affected, which points to this being an issue specifically with *serial* HDF5, as using MPI with HDF5 is unaffected. NetCDF4 is most likely impacted due to it using HDF5 as a storage backend since version 4. When looking at the error bars, which represent the standard error mean, it becomes obvious that performance is suffering, as there are large deviations within times taken to execute a read call.

iteration	**1**	**2**	**3**	**4**	**5**
hdf5-python	40.922s	38.578s	38.871s	506.764s	38.734s
netcdf4-python	478.432s	64.730s	1608.465s	67.262s	162.631s

Table 5.2.: Recorded measurements for hdf5-python (30 GB) and netcdf4-python (50 GB).

When looking at the data up close in Table 5.2, four of ten total iterations are affected by suboptimal performance. For hdf5-python iteration 4 is an outlier due to its $\sim 1280\%$ slower runtime on average compared to all other iterations. For netcdf4-python it is less clear, as $\sim 66\%$ of iterations are above 100 seconds. One could argue that the faster runs could also be caused by caching effects not being properly dealt with.

iteration	1	2	3	4	5	mean	standard deviation
hdf5-python	23.117s	22.511s	22.637s	22.260s	22.333s	22.571s	
deviation	0.546s	−0.061s	0.066s	−0.311s	−0.238s		0.303s
netcdf4-python	53.259s	51.949s	53.838s	53.082s	51.183s	52.662s	
deviation	0.596s	−0.713s	1.175s	0.419s	−1.479s		0.960s

Table 5.3.: Standard deviation for hdf5-python (20 GB) and netcdf4-python (40 GB).

In order to show that this is not the case, results need to be evaluated carefully. When looking at the mean-time taken for "stable" iterations, such as with hdf5-python for a 20 GB file and netcdf4-python for a 40 GB file (Table 5.3) and calculating the *standard deviation*, iterations which are unaffected usually fall within a seconds. Calculating the *relative standard deviation* (RSD) for both shows about a $\sim 1.34\%$ deviation for hdf5-python and $\sim 1.82\%$ for netcdf4-python for these specific file sizes chosen.

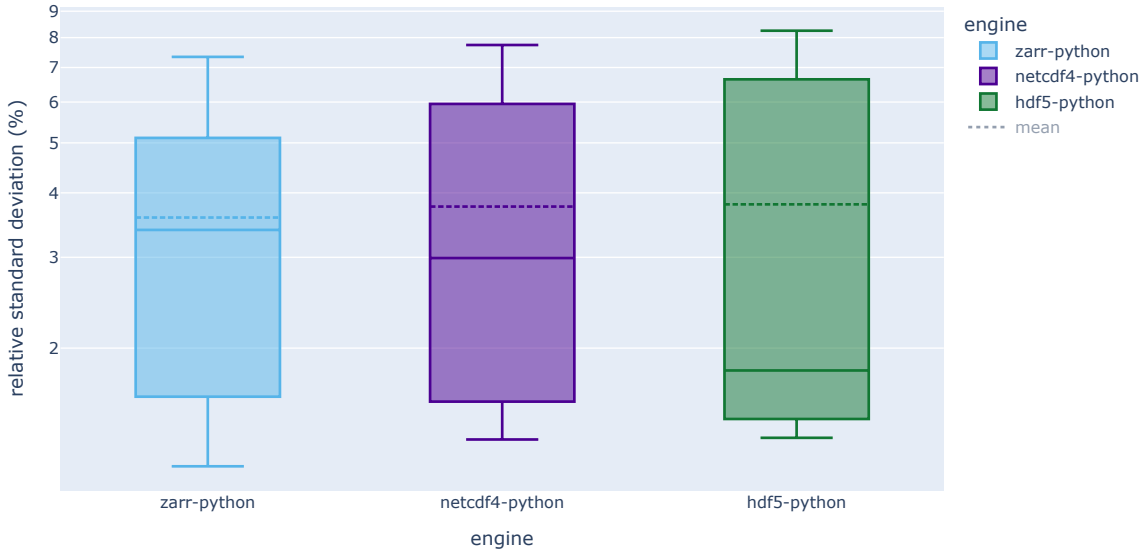


Figure 5.2.: Relative Standard Deviation. Y-Axis is log scale.

When plotting all iterations that are unaffected by these massive deviations in comparison, results are usually within a mean of $\sim 3.7\%$ (Figure 5.2 dotted line) of each other. This will be considered *run-to-run variance* going forward. Zarr was plotted to show the RSD of a format that is not generally impacted under the same circumstances.

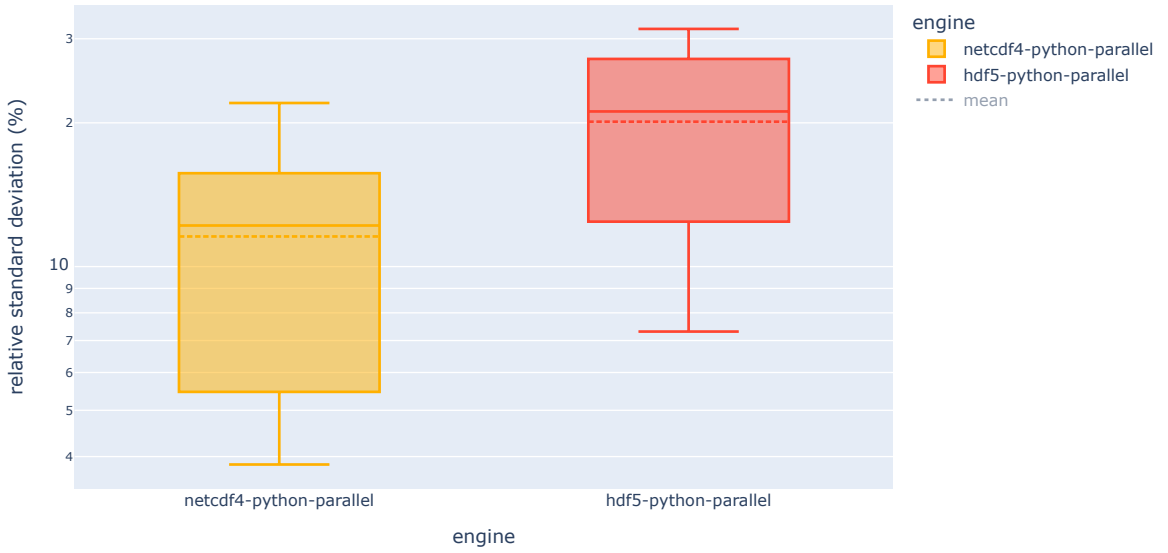


Figure 5.3.: Relative Standard Deviation. Y-Axis is log scale.

Both parallel HDF5 and NetCDF4 have been omitted due to RSD being less representative in their case. Both show significantly higher deviations (Figure 5.3) and thus a higher mean percentage (dotted line) at $\sim 11.56\%$ for NetCDF4 and $\sim 20.11\%$ for HDF5.

iteration	1	2	3	4	5
netcdf4-python-parallel	1.895s	1.686s	1.370s	2.083s	2.041s

Table 5.4.: Recorded measurements for netcdf4-python-parallel (10 GB).

Table 5.4 shows the relative standard deviation at $\sim 14.45\%$ while the actual slowdown is simply a few milliseconds (ms), which may as well be down to a OS scheduling delay or a delay in resource availability. This is due to parallelism being much more susceptible to fluctuations, as both formats levering MPI are much faster at reading the dataset. Therefore even small differences in run-time will represent large deviations percentage wise.

However, coming back to serial HDF5, this does not mean that iteration 4 from Table 5.2 is not indicative of real world performance, since the slower runtime is still consistent should the exact same node be hit again.

iteration	1	2	3	4	5
hdf5-python	437.620s	468.010s	47.156s	463.439s	45.728s

Table 5.5.: Recorded measurements for hdf5-python (40 GB).

When four nodes are used, it is harder to analyze which nodes are actually the ones affected. Therefore when looking at another example of hdf5-python from a rerun of the benchmark

with just one compute node (Table 5.5), three iterations can be observed with suboptimal performance. In comparison these three iterations specifically are within run-to-run variance of each other with a RSD of $\sim 2.93\%$.

This is due to the fact that the exact same node (l10348) was hit for each of those affected iterations, pointing towards consistent performance when an impacted node is used. l10348 is not the only affected node that was found. Multiple other nodes have also been observed (Listing A.4) with no obvious pattern to them that would help narrow down a cause.

Interestingly l10348 does not affect all formats equally. As noted before only specific formats seem to struggle under these conditions. Zarr for example does not exhibit the same erratic behavior even though l10348 has also been used to perform some of Zarrs repetitions in multiple cases during the whole benchmark.

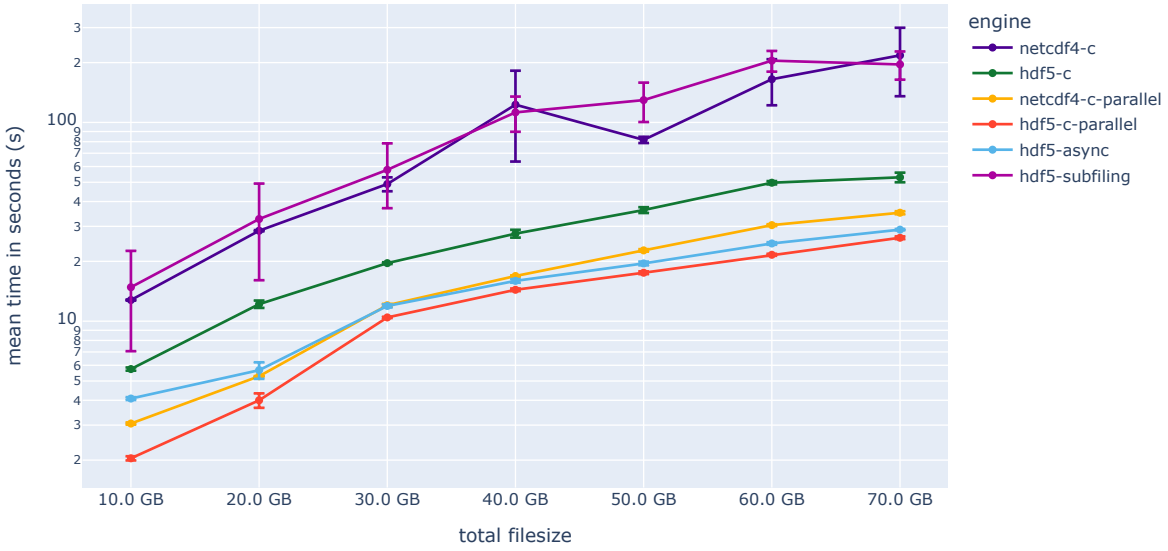
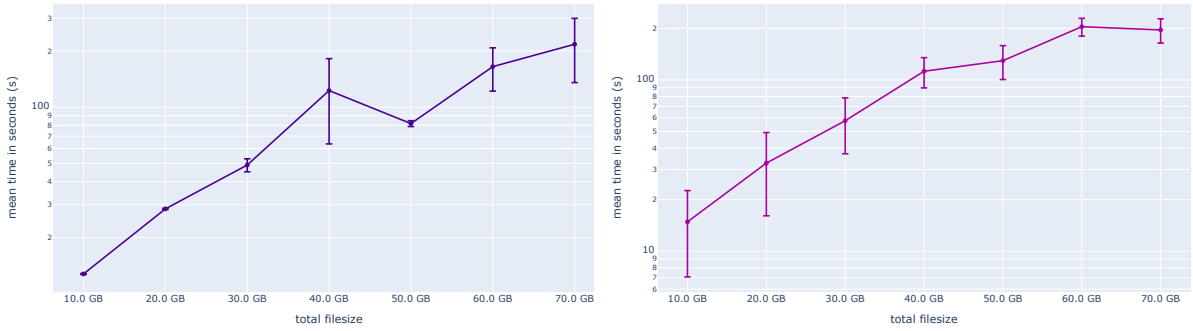


Figure 5.4.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

Additionally Python is not the only programming language suffering from suboptimal performance. C also shows signs of serial cases being impacted, though only for *netcdf4-c* (Figure 5.4). Iterations ranging from 30GB to 70GB are obviously outliers, only exception being 50 GB. Curiously *hdf5-c* is not impacted exhibiting exceptionally stable performance. This could be due to hdf5-c getting "lucky" during node selection and not encountering one of the lower performing nodes.



(a) NetCDF4 (b) Subfiling

Figure 5.5.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

Hdf5-subfiling might be impacted as well, judging by the error bars, representing the first instance of an MPI based feature. When looking at *netcdf4-c* and *hdf5-subfiling* at their own in Figure 5.5a and Figure 5.5b this behavior should be clearly visible.

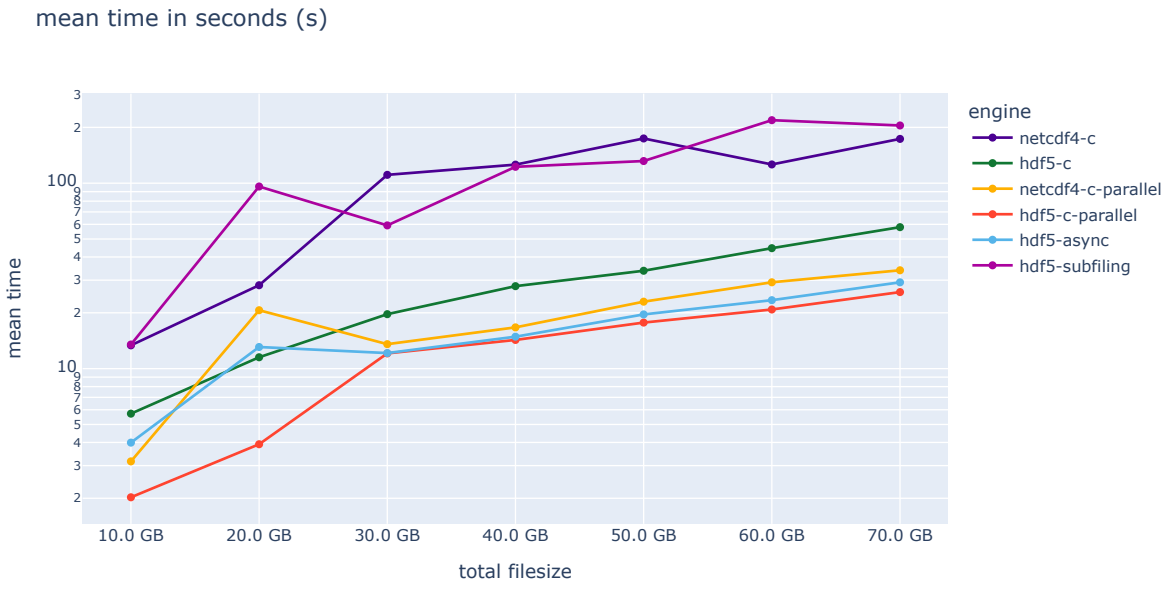


Figure 5.6.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

No additional formats are affected, although there have been instances, where *netcdf4-c-parallel* and *hdf5-async* seemed to also be impacted at 20 GB (Figure 5.6).

The exact reasons for this inconsistent behavior of some the nodes mostly affecting serial HDF5 and NetCDF4 should be investigated further and will be mentioned in Section 6.1. It is however outside the scope of this work.

Going forward results will be showing outliers included, as it has been shown that affected runs are still representative of how nodes on the cluster can perform, should one be “unlucky” and get them assigned to their application.

5.3. Python benchmark

A general overview will be given before dividing the plots into specific features.

5.3.1. General overview

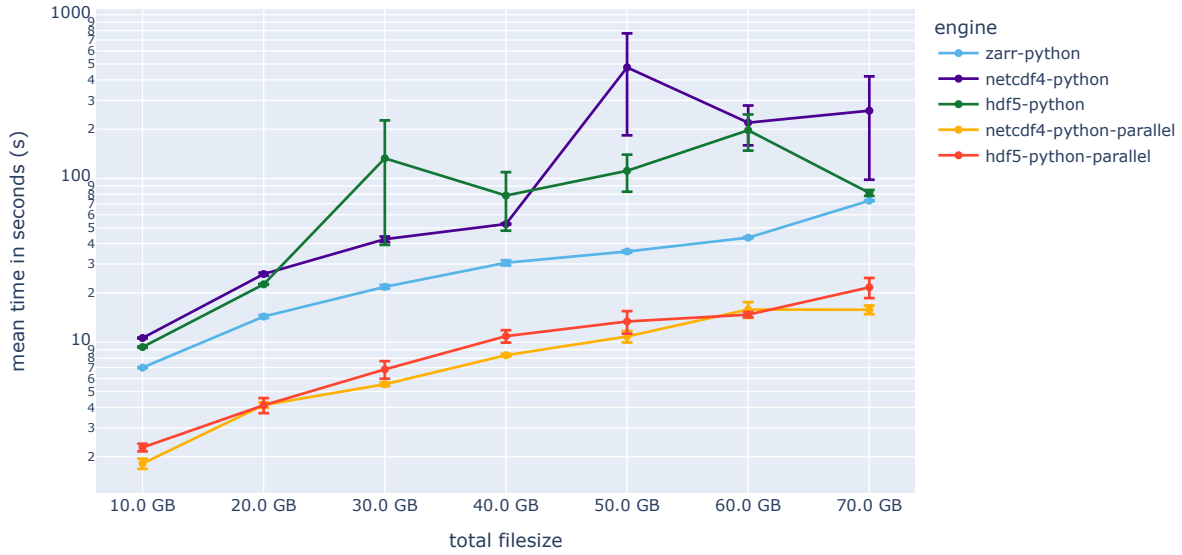


Figure 5.7.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

First looking at a line plot of dataset ranges from 10-70 GB in 10 GB steps, a large gap can be seen of about $\sim 89.50\%$ on average, separating the formats simply relying on serial execution from those that use parallelism via MPI. When removing runs that are affected by the node-phenomenon (3 runs for NetCDF4, 4 runs for HDF5) as detailed in Section 5.2, the gap shrinks to $\sim 71.13\%$ on average. Parallelized workloads being faster is to be expected as more cores / threads partake in reading a dataset, lessening the burden on each of them as they only have to read $\frac{1}{\text{nth}}$ the amount of data that just one core / thread would handle otherwise. Additionally I/O calls are executed collectively theoretically decreasing the number of calls required, making more efficient use of system resources.

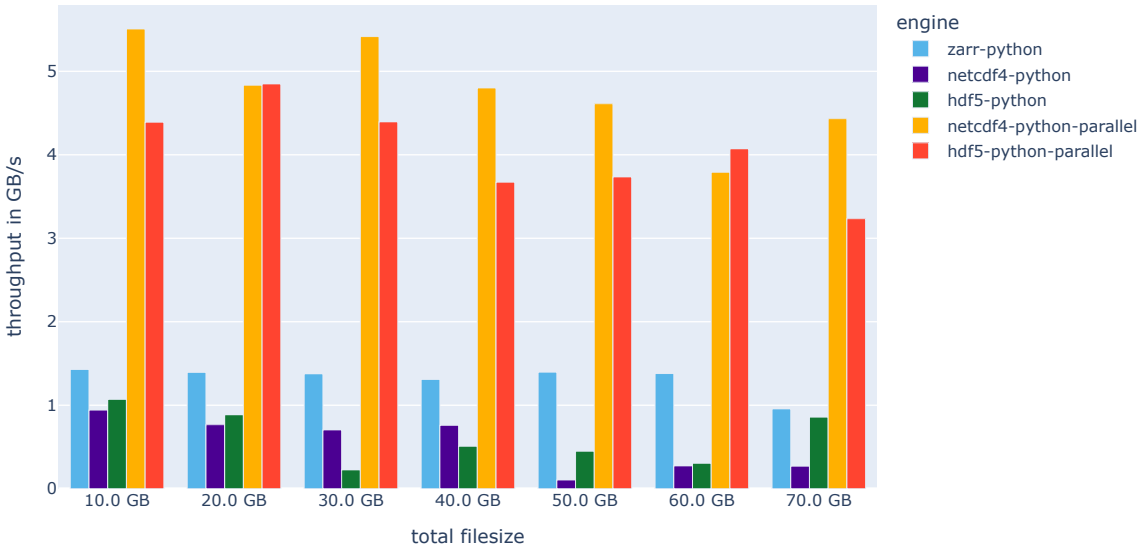


Figure 5.8.: Average throughput per file format and feature used across varying dataset sizes.

When examining throughput of the system, a stark $\sim 81.25\%$ difference is visible, representing a 5.33x increase in throughput on average between serial and parallel workloads. This clearly showcases the advantage properly parallelized formats have. Overall improvement is somewhat exaggerated, since nodes do not perform the same. When accounting for affected runs, as shown in Section 5.2, parallelism is $\sim 75.34\%$ or 4.05x faster.

Serial Python

Figure 5.9.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

Zarr is performing exceptionally well in comparison, when isolating serial measurements. It outperforms NetCDF4 by $\sim 79.19\%$ and HDF5 by $\sim 64.28\%$ on average. When removing affected runs the advantage shrinks to $\sim 44.14\%$ for NetCDF4 and $\sim 16.75\%$ for HDF5.

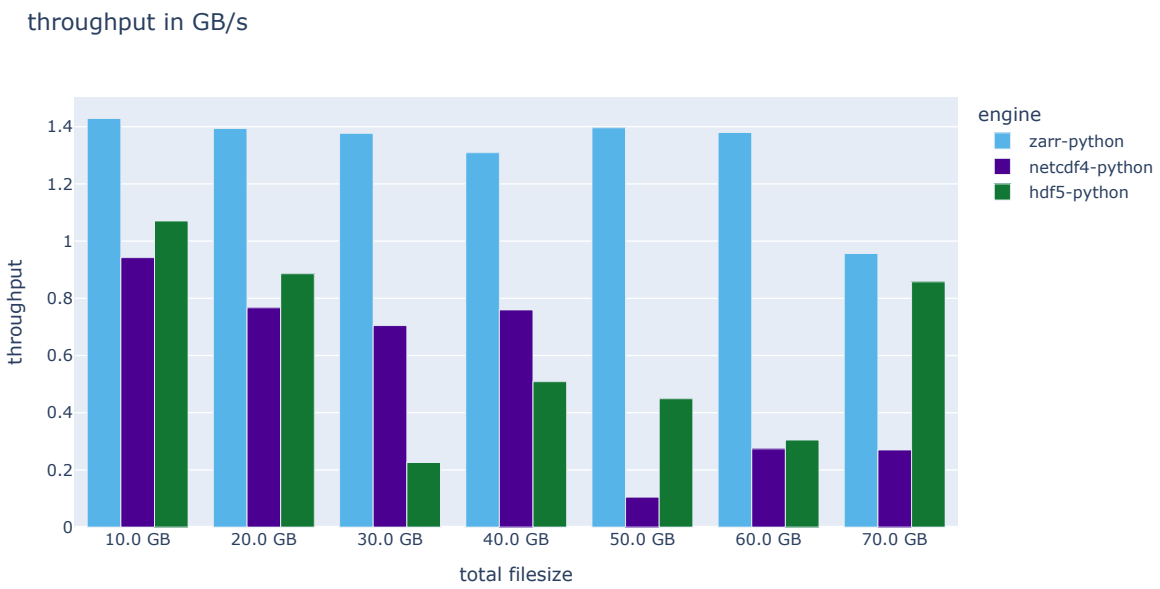


Figure 5.10.: Average throughput per file format and feature used across varying dataset sizes.

Looking at throughput, Zarr can be seen outperforming other formats though the advantage shrinks as dataset size grows. In this instance Zarr is faster by $\sim 58.64\%$ or 2.42x compared to NetCDF4 and $\sim 53.44\%$ or 2.15x compared to HDF5 on average, which comes down to $\sim 42.39\%$ or 1.74x for NetCDF4 and $\sim 25.54\%$ or 1.34x for HDF5 once outliers are filtered out.

Parallel Python

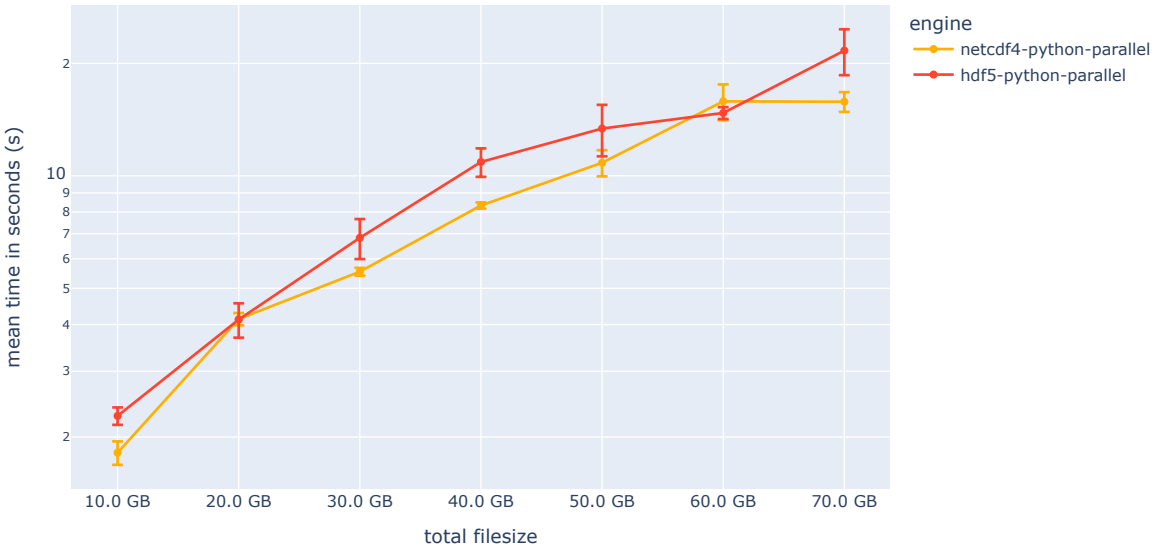


Figure 5.11.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

When isolating features using MPI, NetCDF4 is generally faster by $\sim 15.70\%$, which does represent a sizable improvement. Since there seem to be no anomalies with MPI based features in Python, no additional adjustments need to be made to account for them.

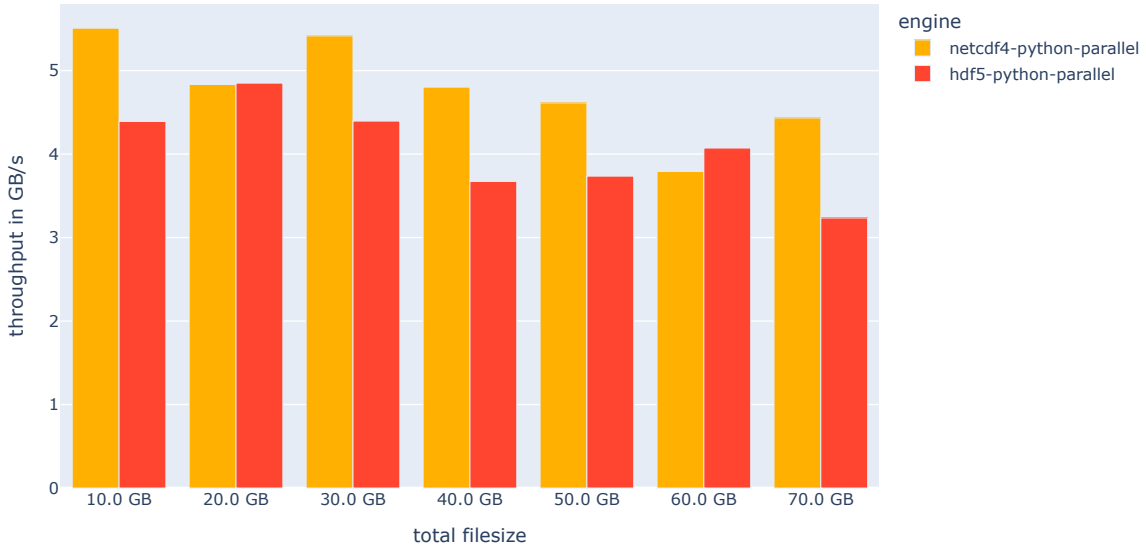


Figure 5.12.: Average throughput per file format and feature used across varying dataset sizes.

For throughput, trending downwards with filesize, NetCDF4 offers 15.11% better performance or a 1.18x improvement.

5.3.2. Comparison using different chunksizes

Moving onto chunked datasets as mentioned in Section 2.8.1, comparisons will be made using different chunksizes for a 10GB *float64* dataset with a value of *[None None]* representing the full file without any chunking as a baseline. Zarr is the only exception as it does auto-chunking by default and does not "do" contiguous layouts directly. It is possible to force Zarr to "act" like it is storing data contiguously, by creating a dataset that is effectively one chunk, but that would counteract Zarr's whole philosophy to handling data and is therefore unrealistic. Additionally it should be noted that the data should not inherently benefit from chunking, nor is any specific pattern used that would be helped with dividing data into chunks. This means that the results serve as a comparison point for how the formats behave for different chunksizes, not if the data within specifically benefits from being accessed in a chunked layout.

Serial Python

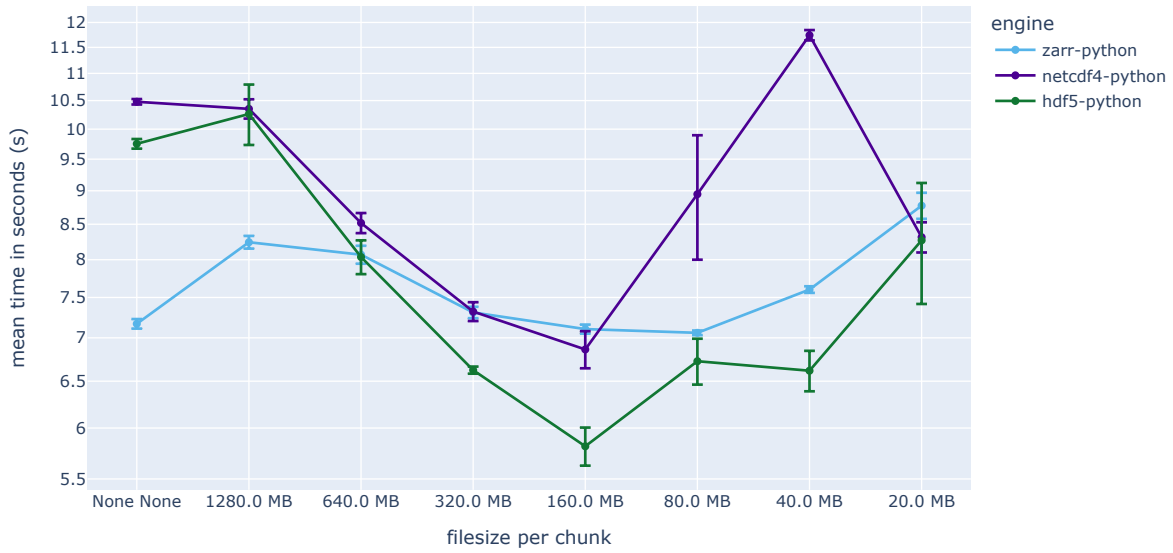


Figure 5.13.: Loading a file with one variable with varying chunksizes sizes for a 10 GB file. Y-Axis is log scale.

An optimum (Figure 5.13) is reached with a chunksize of 160 MB for both NetCDF4 and HDF5. Decreasing the chunksize further has no benefit and seems to actively hurt performance. Zarr scales well down to 80 MB with a very slight improvement from 160MB, which is within run-to-run variance. Compared to Zarr's base case, which does have auto-chunking enabled setting chunks to be 5 MB, results are almost identical showing that Zarr is able to determine decent chunksizes to use. This enforces the need for careful consideration when choosing a chunksize to use. Misconfiguration can negatively affect performance as for example with NetCDF4 at 40 MB and basically all potential sizes for Zarr, except 80 and 160MB.

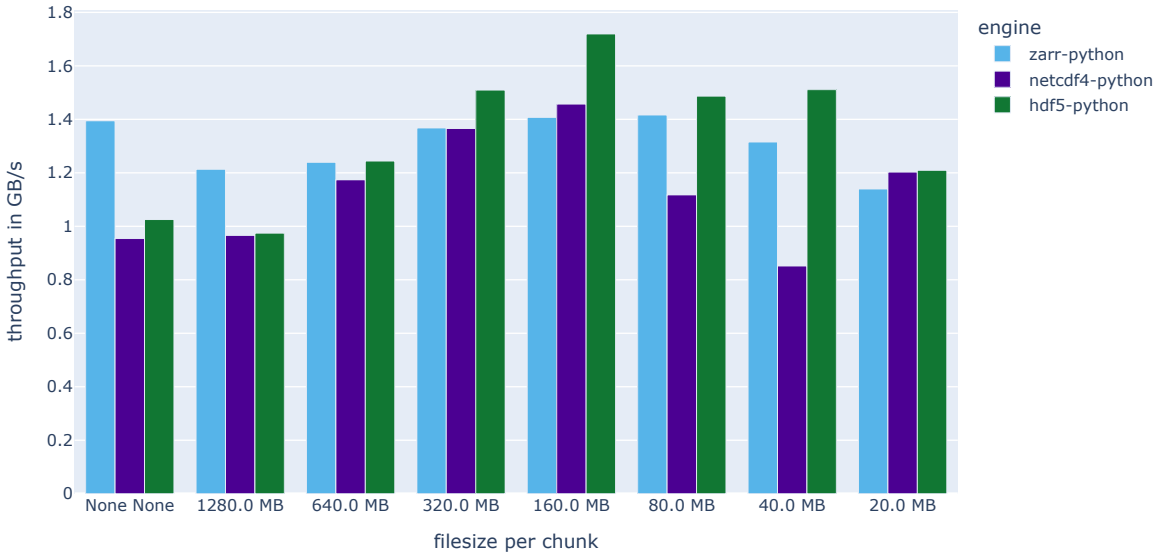


Figure 5.14.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

There is one more important observation, when looking at throughput (Figure 5.14), which might not be immediately obvious in regards to the maximum performance achieve by HDF5 in the case of a 160 MB chunksize. Herein lies the first instance where a software bottleneck might have been encountered on Levante. As noted in Section 5.1 maximum throughput using Lustre for single stream performance is usually in range of 1-3.2 GB/s, which could potentially have been hit with HDF5 reaching a peak of 1.72 GB/s.

Taking some measurements with IOR[8], a commonly used I/O benchmark in HPC, single stream performance is shown to be about 3.98 GB/s, using just a simple configuration that creates a 10 GB file on a single node using a single task.

[8] `https://github.com/hpc/ior`

Parallel Python

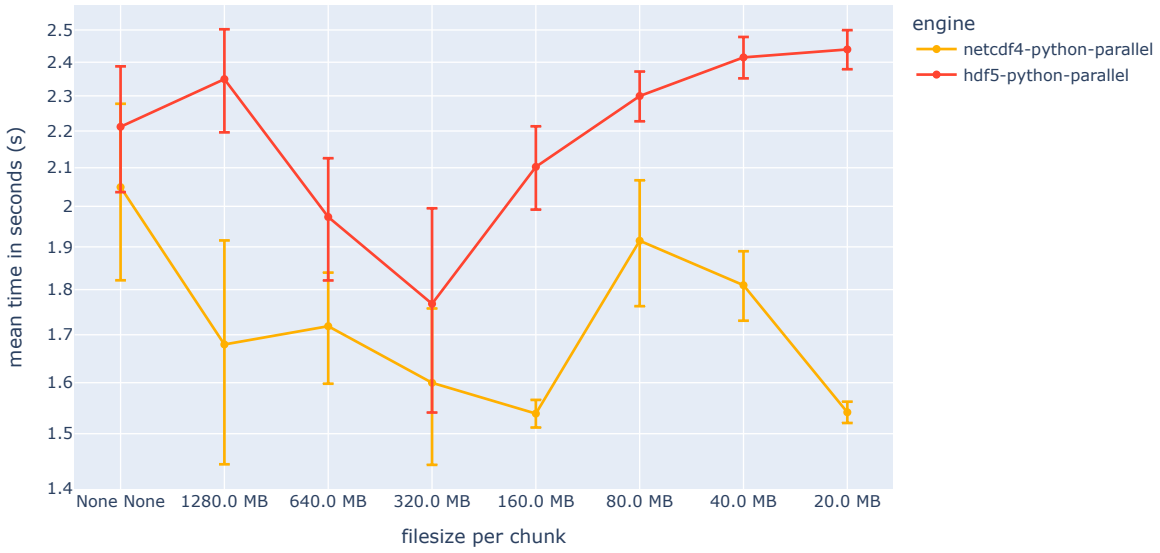


Figure 5.15.: Loading a file with one variable with varying Chunksizes sizes for a 10 GB file. Y-Axis is log scale.

For formats using MPI the picture varies somewhat as HDF5 actually performs best at 320 MB, while NetCDF4 lines up with the previously discussed serial examples at 160 MB.

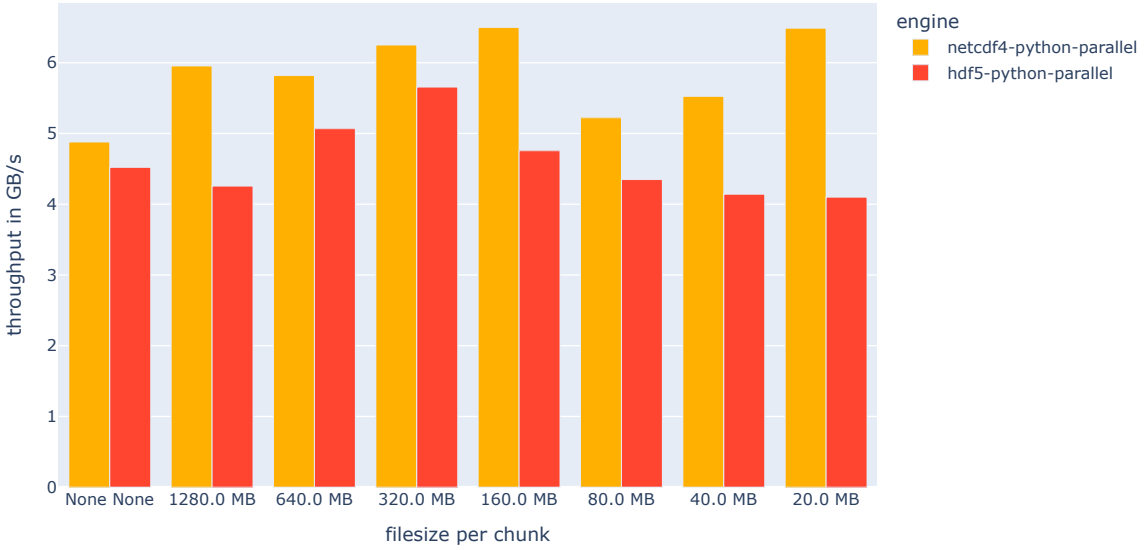


Figure 5.16.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

Chunking does improve performance greatly, allowing NetCDF4 to hit 6.49 GB/s up from 4.87 GB/s representing a 1.33x improvement, while HDF5 is 1.25x faster.

5.4. C benchmark

Looking at the C interface for some formats, a general overview will be given, dividing plots in a similar manner to the previous Sections covering Python. It should be noted that in cases with parallelism, features will be present that are not supported by all formats listed. Specifically async and subfiling, which are not supported by NetCDF4, indicated by an appropriate naming scheme.

5.4.1. General overview

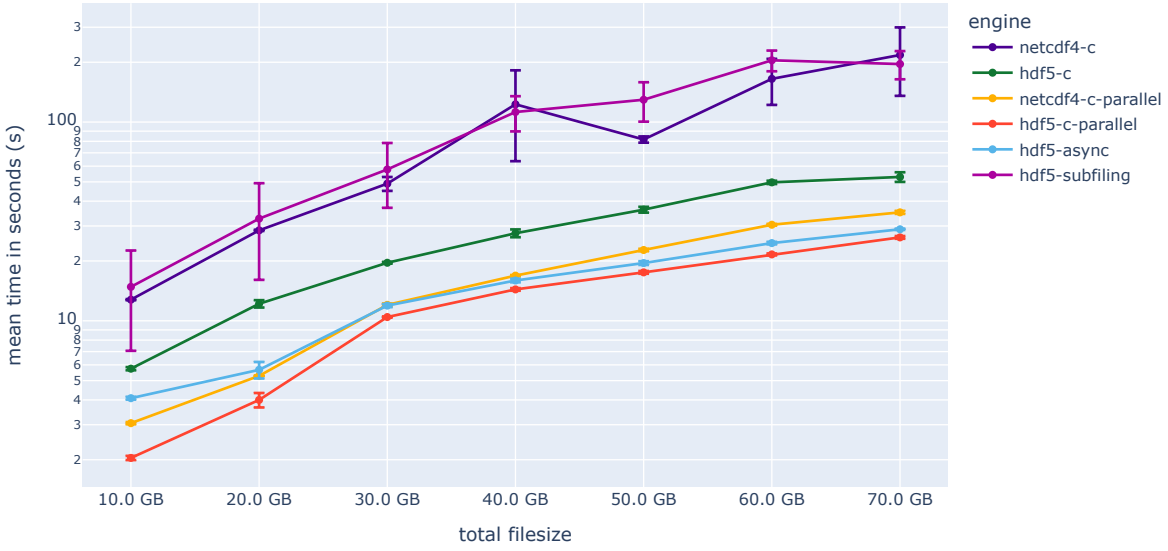


Figure 5.17.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

Similar to Python, parallelism outperform both serial formats (Figure 5.17), though curiously *hdf5-subfiling* performs almost the same as serial *netcdf4-c*. For a format requiring MPI and therefore using 32 Ranks this is quite surprising. This does heavily affect the average when comparing parallel to serial, with parallel bringing only a $\sim 38.73\%$ improvement. When removing subfiling from the equation the gap widens to $\sim 74.85\%$. Though this also includes some suboptimal runs for *netcdf4-c* (4 in total), when adjusting for those runs as well the average is about $\sim 51.59\%$.

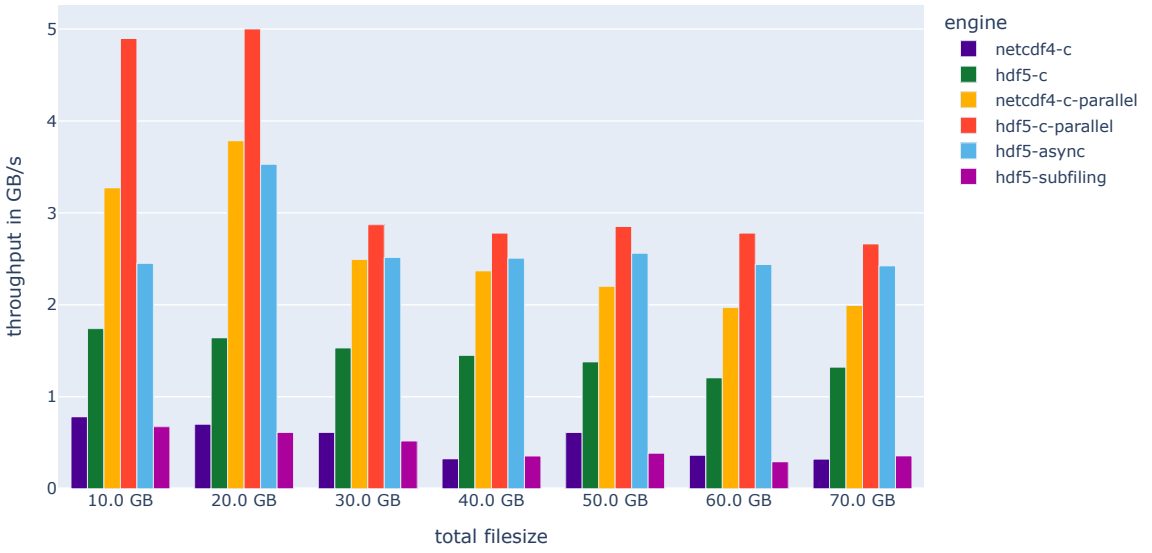


Figure 5.18.: Average throughput per file format and feature used across varying dataset sizes.

Looking at throughput again, a similar picture is showing with parallel offering a $\sim 55.97\%$ or 2.27x improvement over serial on average, while this gap grows to $\sim 65.22\%$ or 2.88x without *hdf5-subfiling* and adjusts downwards to $\sim 56.96\%$ or 2.32x when also excluding the affected runs for *netcdf4-c*.

Serial C

Figure 5.19.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

Just focussing on serial cases (Figure 5.19), HDF5 is on average $\sim 69.88\%$ faster than NetCDF4. This is again due to some runs being affected by anomalies. When adjusting for them the gap shrinks to $\sim 55.97\%$ still giving the edge to HDF5.

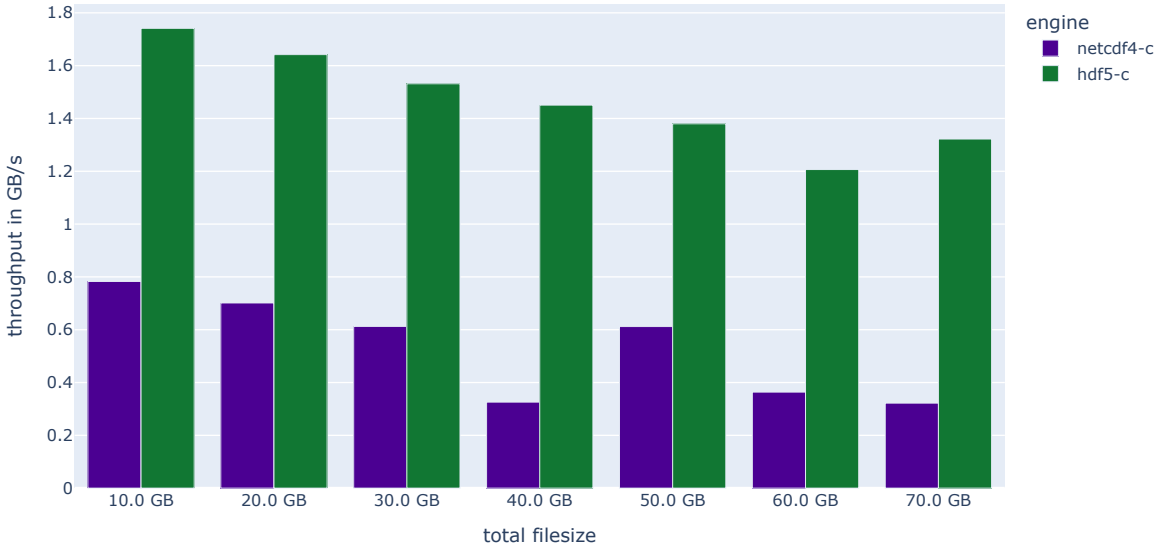


Figure 5.20.: Average throughput per file format and feature used across varying dataset sizes.

For throughput, serial HDF5 does seem to be bound by driver overhead as dataset sizes increase. HDF5 achieves 1.74 GB/s for a 10 GB file, putting it in the range where HDF5 could potentially be limited by Lustre as mentioned with Python previously.

Parallel C

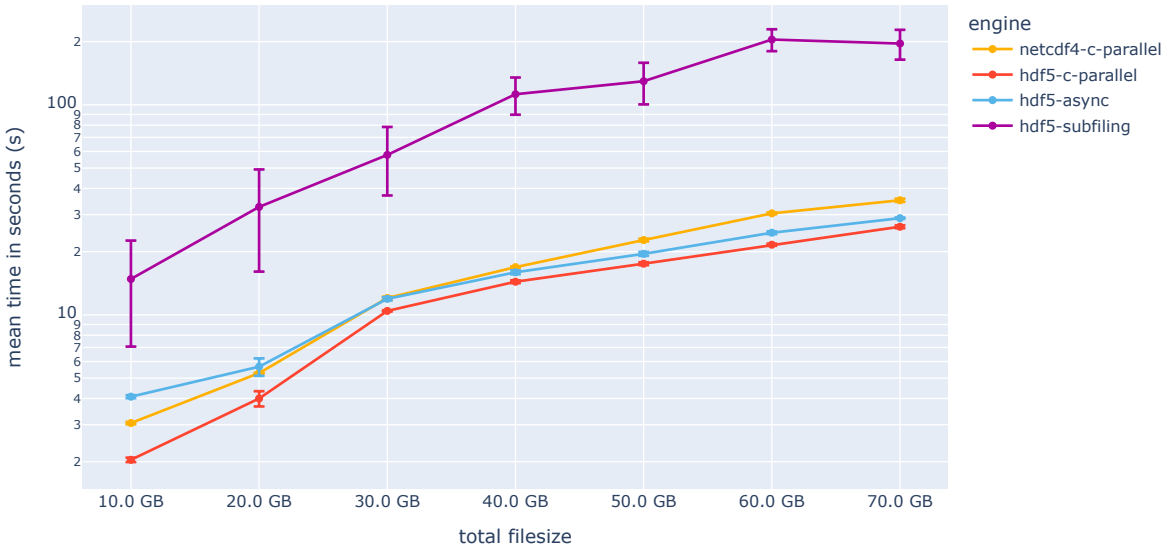


Figure 5.21.: Loading a file with one variable with varying dataset sizes. Y-Axis is log scale.

MPI based features are similar in that *hdf5-c-parallel* outperforms all other formats. Both NetCDF4 and *hdf5-async* are close relative to one another, only showing a larger gap at 10 GB. It should be noted that the read scenario portrayed by the benchmark, does not include any form of computation involving the data being read. While *hdf5-async* is falling behind *hdf5-c-parallel*, which performs synchronous I/O calls, async should take the lead once compute tasks are introduced, since asynchronous I/O can overlap I/O calls with computation being performed.

Hdf5-subfiling should not be used with the current configuration as it offers the worst performance while being the most complex. In regards to Section 3.2, subfiling should be able to offer better performance, but does seem to require a much higher number of ranks to facilitate that scaling.

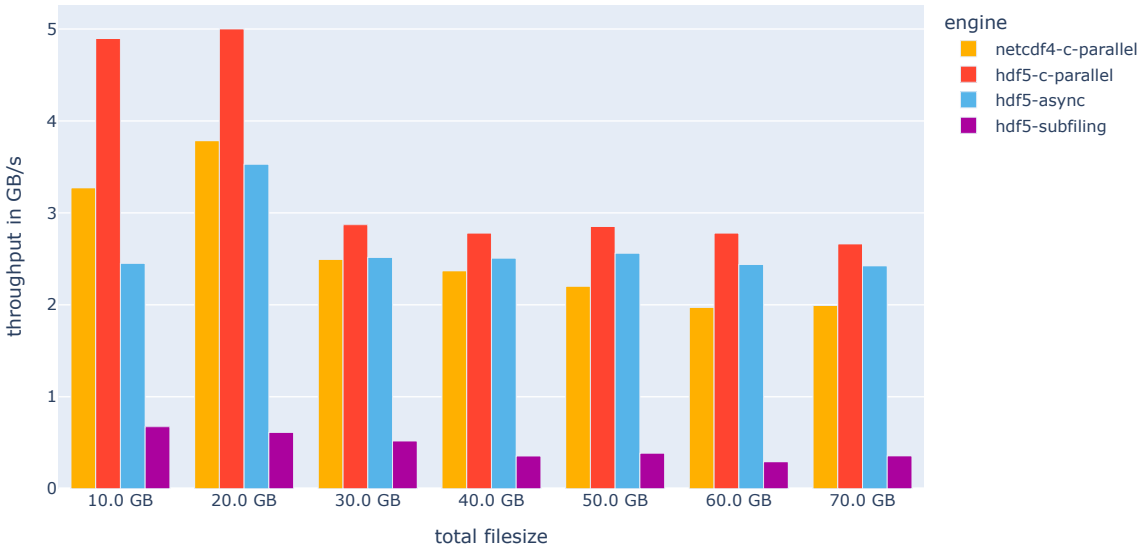


Figure 5.22.: Average throughput per file format and feature used across varying dataset sizes.

The performance difference between *netcdf4-c-parallel* and *hdf5-c-parallel* specifically (Figure 5.22), also is consistent to Bartz et al., as mentioned in Section 3.3, with *hdf5-c-parallel* generally outperforming *netcdf4-c-parallel*. *Hdf5-async* does show the higher overhead of asynchronous I/O again, which should be accounted for when trying to match features to use-cases.

5.4.2. Comparison using different Chunksizes

Evaluating chunked datasets with their respective chunksizes, a 10 GB file filled with *float64* elements will be the baseline again denoted as *[None None]*, which always represents a contiguous layout.

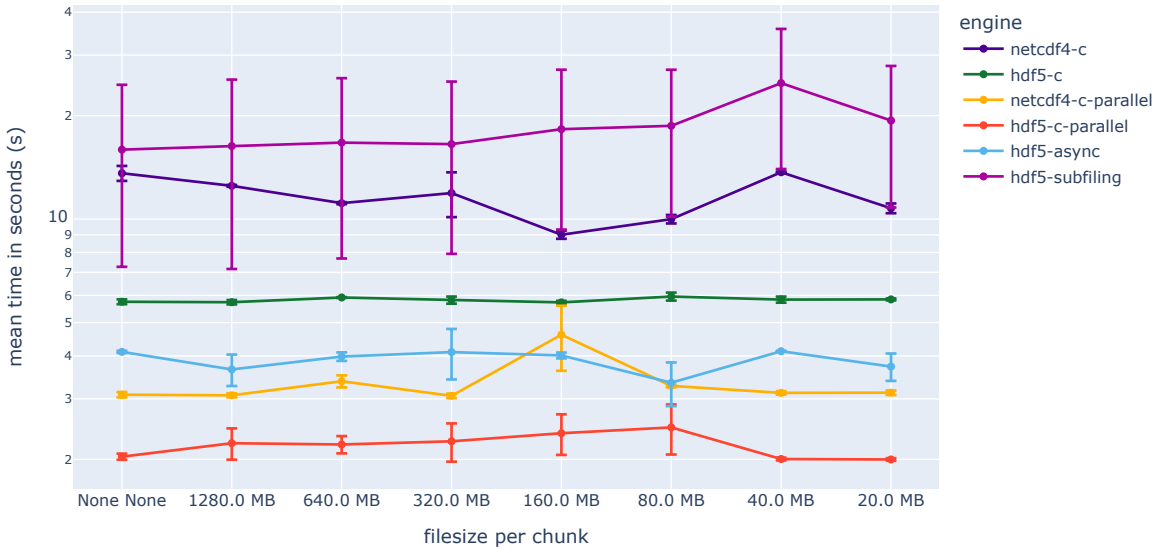


Figure 5.23.: Loading a file with one variable with varying Chunksizes sizes for a 10 GB file. Y-Axis is log scale.

Figure 5.23 shows a quite contrasting picture compared to Python, with formats mostly offering no or minimal improvements. The only exception being *netcdf4-c*, which does seem to benefit somewhat. This could mean that the example chosen offers minimal opportunity for chunking to improve performance.

Serial C

Figure 5.24.: Loading a file with one variable with varying Chunksizes sizes for a 10 GB file. Y-Axis is log scale.

When looking at the serialized default case first in Figure 5.24, HDF5 is still ahead of NetCDF4 and does not seem to benefit at all from chunking. NetCDF4 improves overall, seemingly hitting the same optimum as its Python counterpart at 160 MB offering $\sim 33.77\%$ better performance than baseline, with an additional $\sim 16.28\%$ drop at 20 MB. At 40 MB, NetCDF4 performs worse than with a contiguous layout, showcasing the need for careful consideration to find an optimal chunksize.

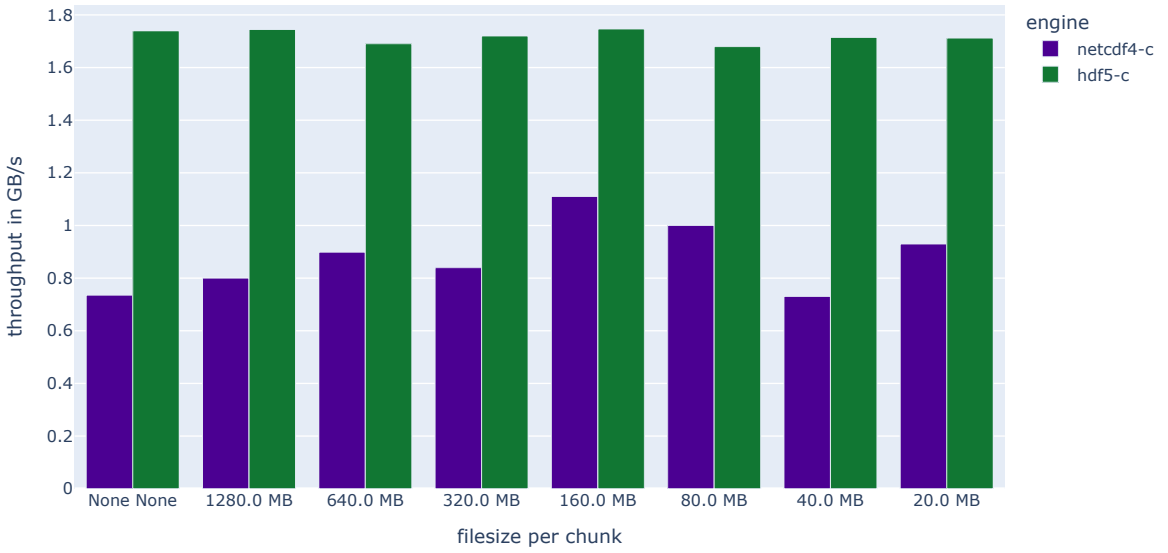


Figure 5.25.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

This behavior can also be seen when looking at total throughput (Figure 5.25). This could potentially be due to hitting the same software bottleneck as discussed with Python previously. There might be additional scaling for HDF5 with different chunksizes, but due to a Lustres limitation no actual improvement is seen in this instance, instead reaching the maximum at 1.746 GB/s, similar to HDF5 in Python.

Parallel C

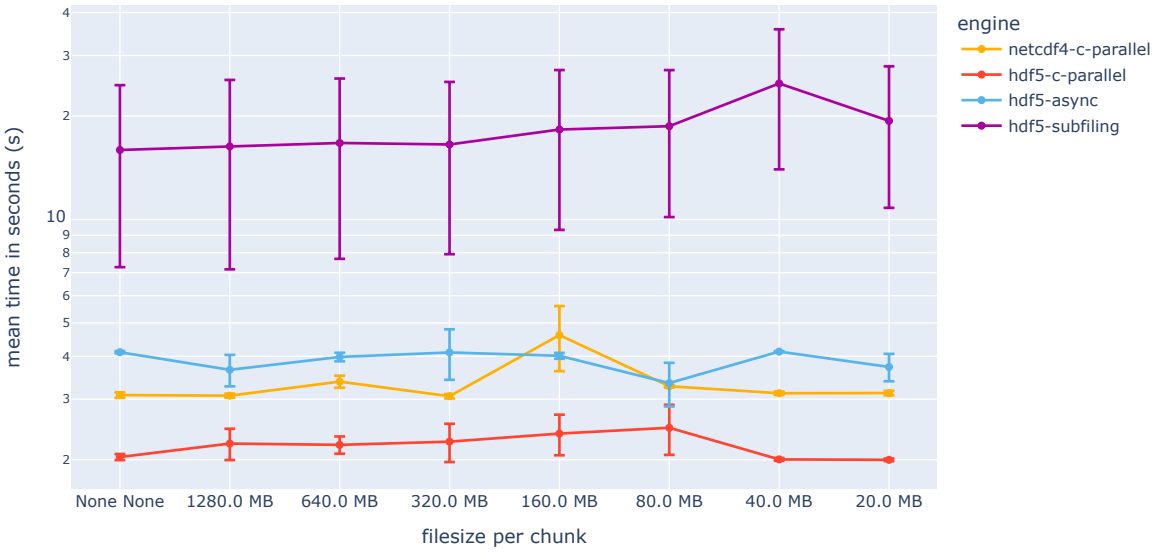


Figure 5.26.: Loading a file with one variable with varying Chunksizes sizes for a 10 GB file. Y-Axis is log scale.

Switching over to features using MPI for parallelism (Figure 5.26), there seem to be minimal improvements. Only *hdf5-async* did improve by $\sim 18.54\%$ for a 80 MB chunksize, while other formats stay at about the same level of performance when compared to the contiguous layout.

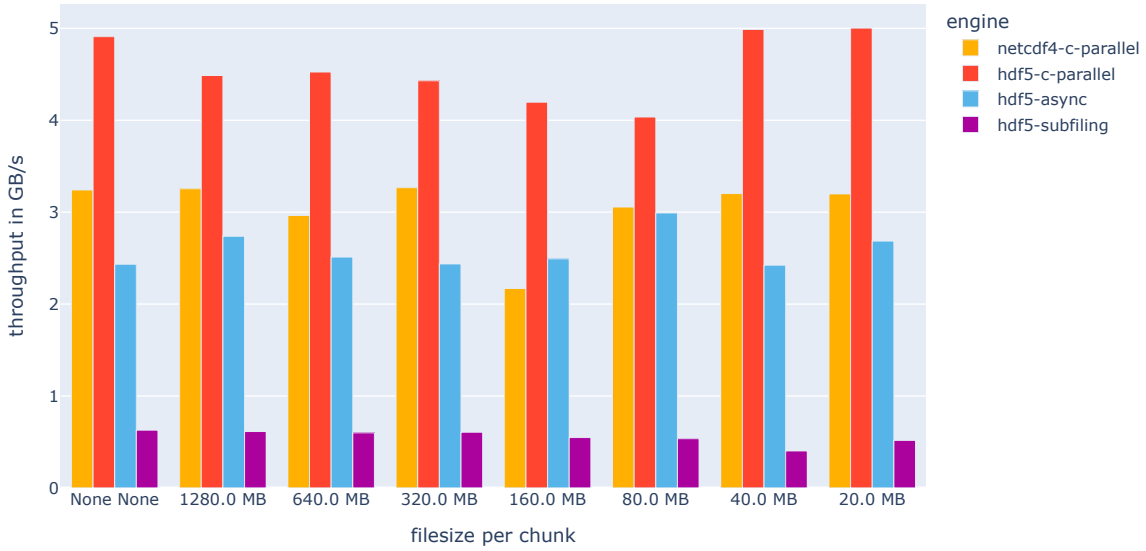


Figure 5.27.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

However, when looking at Figure 5.27 some small improvements can also be seen with *hdf5-c-parallel* allowing it to reach 5 GB/s with a 20 MB chunksize.

5.5. Comparison between C and Python

5.5.1. General overview

Serial baseline

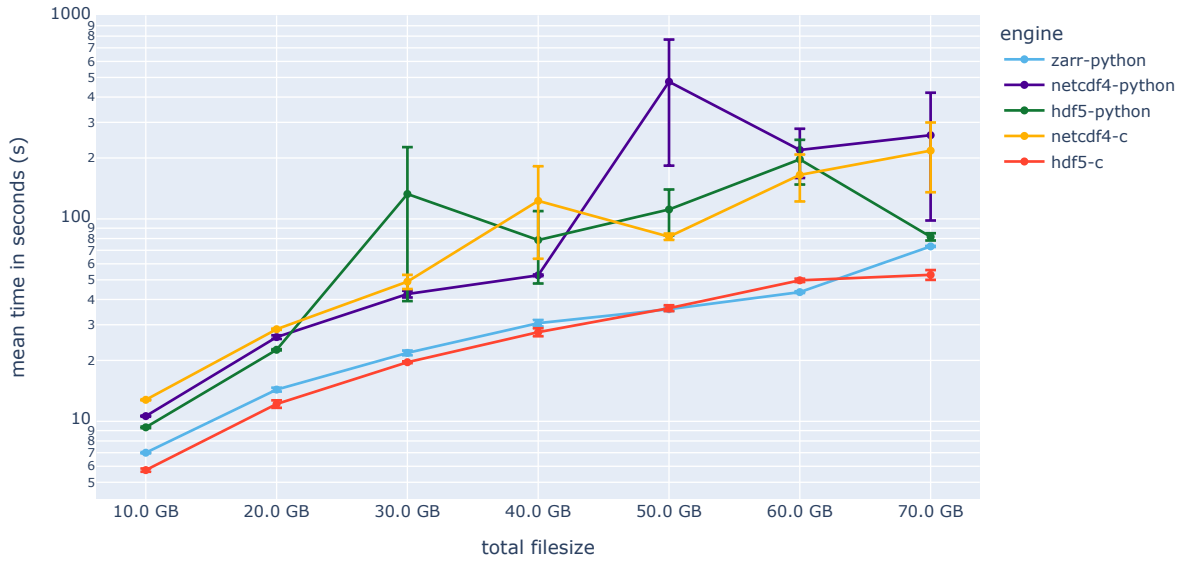


Figure 5.28.: Loading a file with one variable with varying Chunksizes sizes for a 10 GB file. Y-Axis is log scale.

Finally comparing Python and C APIs directly (Figure 5.28). Note the colors do not always align as before. While *netcdf4-python*, *hdf5-python* and *netcdf4-c* mostly perform similar to each other, additionally being the only formats affected by the node anomalies, both *zarr-python* and *hdf5-c* are notable as they perform best, with *hdf5-c* slightly taking the lead. In this instance *hdf5-c* is about $\sim 9.79\%$ faster than *zarr-python* on average.

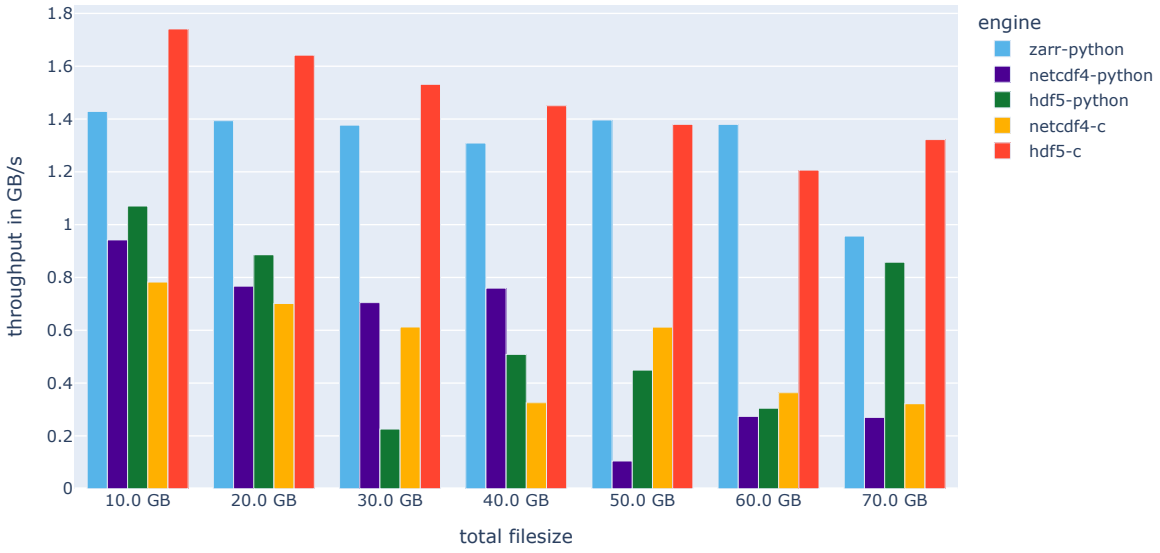


Figure 5.29.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

Evaluating throughput (Figure 5.29) *hdf5-c* is $\sim 10.04\%$ ahead on average, embodying a 1.11x improvement over *zarr-python*.

Parallelized features

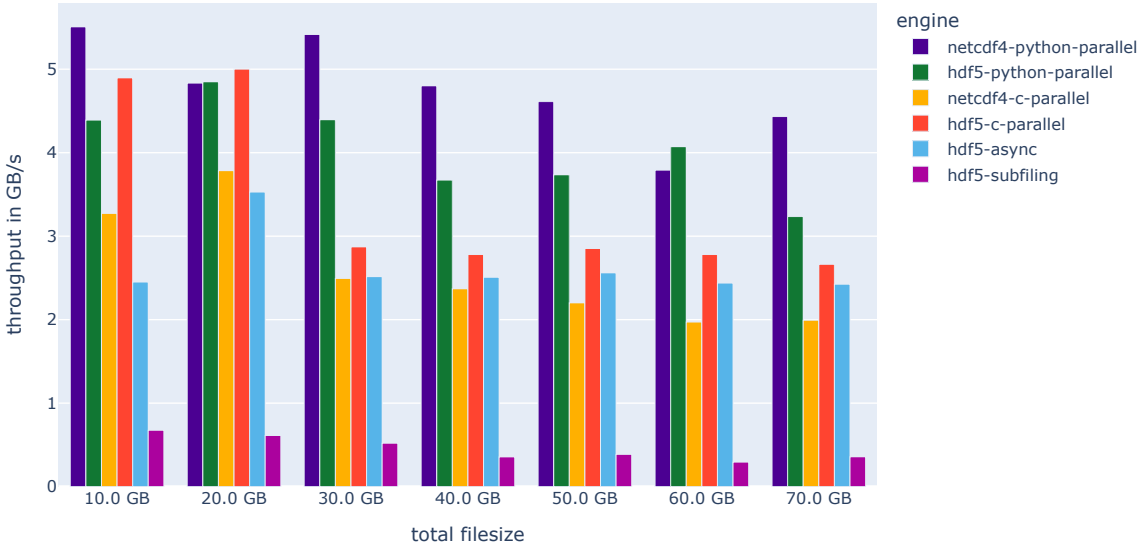


Figure 5.30.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

MPI based features show a somewhat cluttered view as *hdf5-subfiling* distorts the picture when looking at the usual line plot. Therefore, evaluating throughput, *netcdf4-python-parallel* offers the best performance compared to other formats, being $\sim 33.59\%$ faster on average (dropping *hdf5-subfiling*). The next best performing format is *hdf5-c-parallel*, which does manage to surpass *netcdf4-python-parallel* for a 20 GB file, though *netcdf4-python-parallel* is usually $\sim 28.60\%$ faster across the remaining file sizes.

5.5.2. Comparison using different Chunksizes

Serial baseline

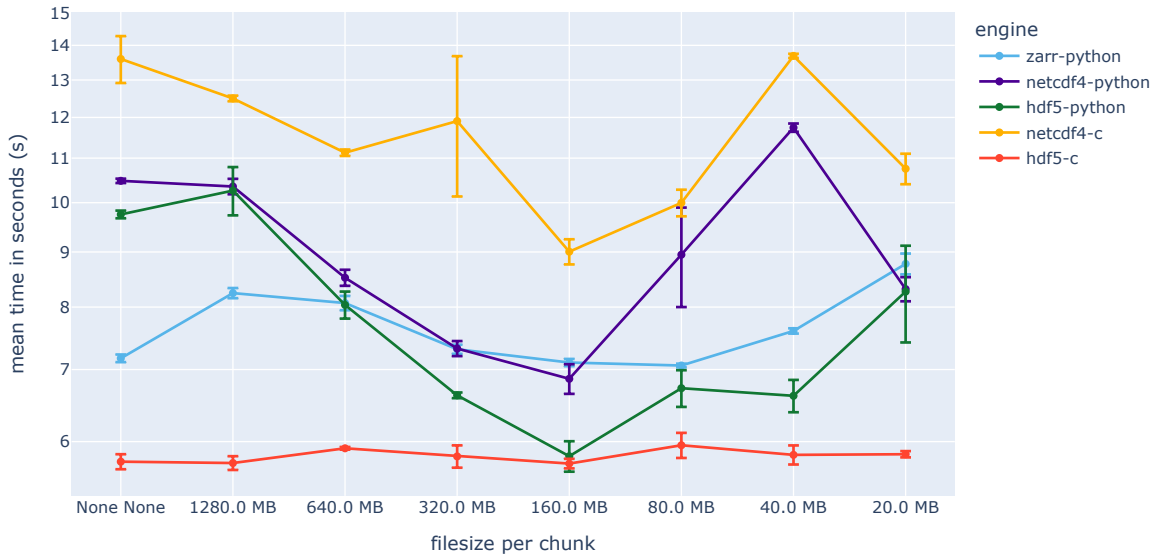


Figure 5.31.: Loading a file with one variable with varying Chunksizes sizes for a 10 GB file. Y-Axis is log scale.

When just looking at the line plot (Figure 5.31), as it offers the best visibility for notable changes, *hdf5-python* is almost able to match *hdf5-c* at 160 MB. This could serve as another indicator that the limitation in Lustre is most likely being hit. Additionally, when looking at both versions of NetCDF4 in either C and Python, a similar line trend is visible, except for a single outlier, pointing towards consistent behavior in either programming language with better scaling achieved by Python. Since NetCDF4 uses HDF5 as its storage backend, this could mean *hdf5-c* might also be consistent with *hdf5-python* in its scaling. Under this assumption *hdf5-c* could potentially offer better performance with chunking enabled, but simply is not able to in this environment, due to the buffered I/O bottleneck in Lustre.

Parallelized features

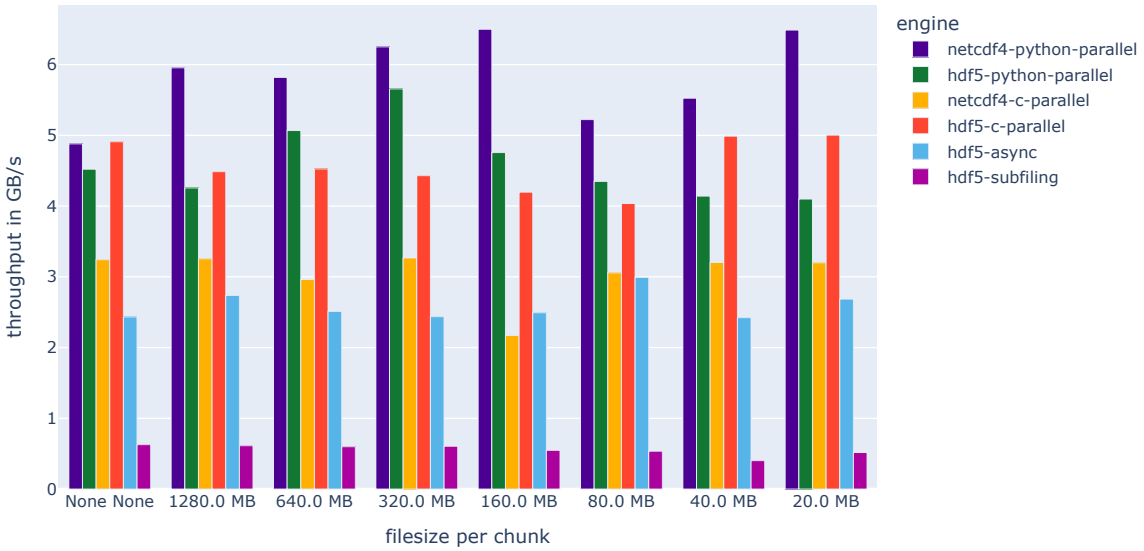
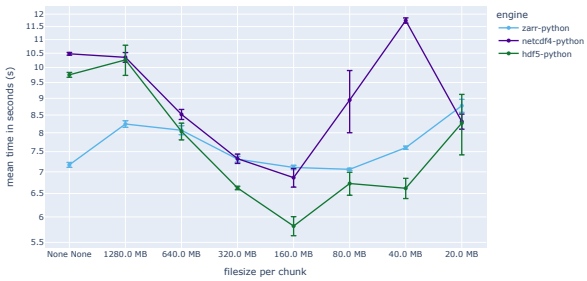


Figure 5.32.: Average throughput per file format and feature used across varying chunksizes for a 10 GB file.

Finally looking at Figure 5.32 *netcdf4-python-parallel* is performing the best, being able to extends its lead given an optimal chunksize.

5.6. Summary of results

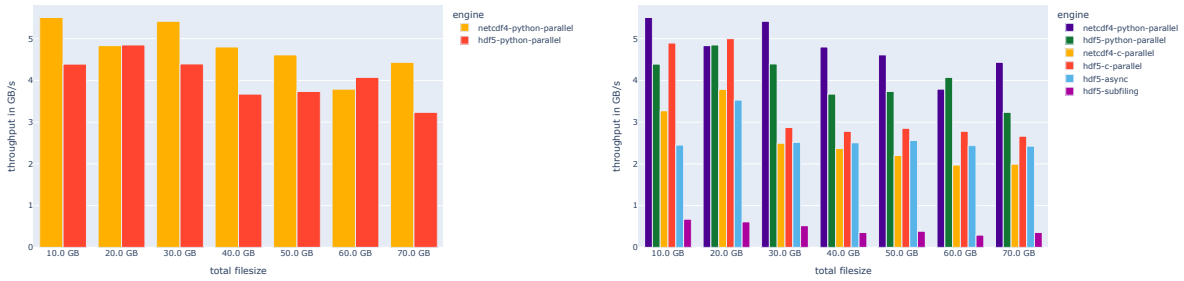
(a) Python, serial & contiguous



(b) Python, serial & chunked, 10 GB

Figure 5.33.: Summary overview of Python formats

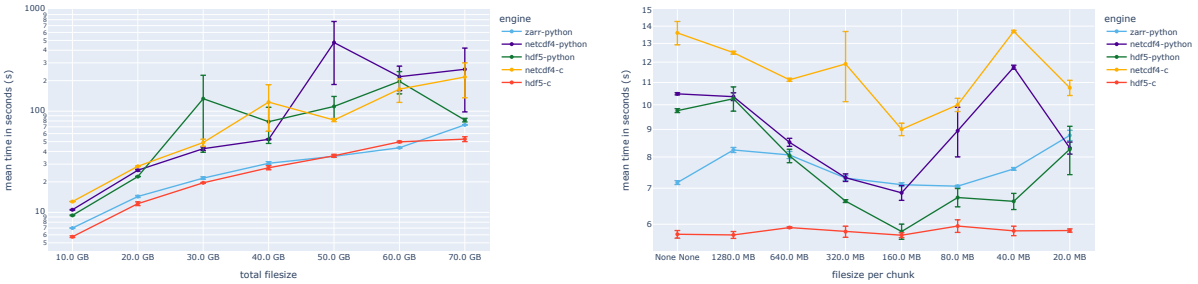
Starting with serial Python, results are clearly in favor of Zarr when no chunking is used (Figure 5.33a), with Zarr offering superior performance to both HDF5 and NetCDF4. In regards to chunking (Figure 5.33b), Zarr quickly gets outperformed by HDF5. Switching between API's could be an option for developers looking to optimize performance, as they are simpler by design and fairly similar in Python, with less setup being required in general.



(a) Python, parallel & contiguous (b) Python & C, parallel & contiguous

Figure 5.34.: Summary overview of Python & C formats

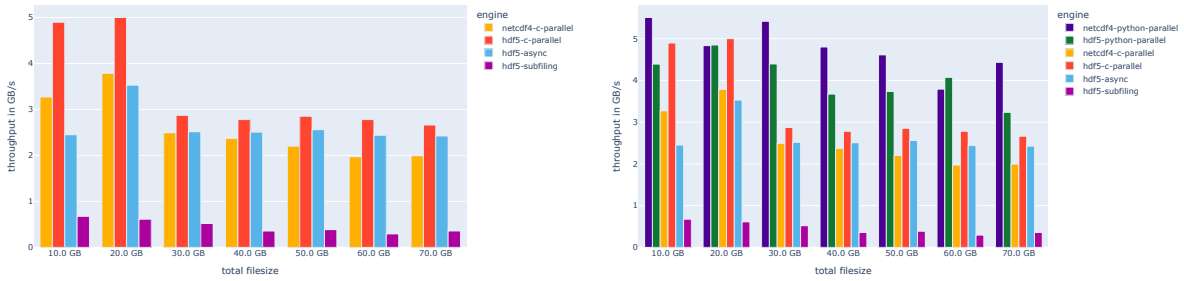
Parallelization clearly favors NetCDF4 in Python (Figure 5.34a), even when compared to C (Figure 5.34b), offering the best performance during the benchmark overall. With the picture being no different leveraging chunking, where NetCDF4 is able to hold a lead over the other parallelized formats as well.



(a) Contiguous layout (b) Chunked layout, 10 GB

Figure 5.35.: Summary overview of Python & C formats

For C, HDF5 does have a small advantage for serialized tasks (Figure 5.35a), technically outperforming Python formats here. Yet the performance might not justify the added complexity here, as Zarr is usually within a couple of seconds of *hdf5-c*, considering Python being easier to implement and use, as shown in Section 4.2. The picture is less clear for chunking (Figure 5.35b), due to the fact that the results are less granular, as only 10 GB were tested. *Hdf5-c* offers the best performance with no additional scaling achievable by chunking the data, while *Hdf5-python* can reach parity at 160 MB, which might be due to a limitation in Lustre.



(a) C, parallel & contiguous

(b) Python & C, parallel & contiguous

Figure 5.36.: Summary overview of Python & C formats

For the parallelized C formats, *hdf5-c-parallel* provides the best performance for the scenario tested (Figure 5.36a). Yet it offers little to no benefit when directly compared to the Python counterparts (Figure 5.36b). *Hdf5-c-parallel* can occasionally outperform *hdf5-python-parallel* (Figure 5.30) at smaller file sizes, but it is debatable if the added performance benefits justify the higher complexity of writing C. Choosing C over Python seems mostly dependent on use-case at the current stage, as *hdf5-async* for example might be desirable to facilitate asynchronous I/O, which is not offered in the current Python implementation of HDF5. When considering raw performance, Python NetCDF4 is superior to all other formats, regardless of chunked or contiguous layout. Python specifically has another advantage in that interfaces are fairly similar as shown in Section 4.2. This means that switching formats should theoretically be much easier than with C as well, regardless of the improved performance characteristics during this benchmark.

Overall the results showed clear benefits for formats built in Python, with NetCDF4 outperforming all equivalent formats in C and HDF5 being mostly ahead, especially at larger file sizes. When looking at the scientific community as a whole, most of their work is already done in Python and with formats generally performing better than with C the advantages are clear. Lower complexity, faster code and a broader range of available resources, especially online, clearly work in favor of the adoption of these formats in Python. Judging by the fact that both HDF5 and NetCDF4 are natively made available by Unidata and the HDF Group, shows they are aware.

Chapter 6.

Conclusion

As software changes, so do its characteristics. Bugs are fixed, processes optimized and new concepts developed to offer an evermore performant experience. This can be facilitated throughout different programming languages and interfaces. To take full advantage of these changes and achieve the best possible performance available, different solutions should be evaluated regularly. This is especially true for file formats, as they are usually used in areas where every bit of performance matters such as HPC. As resources are limited, they should not be wasted.

This thesis seeks to evaluate three different formats namely Zarr, NetCDF4 and HDF5 against one another and find common similarities, as well as differences, which can help choose a format for a specific use-case. A custom benchmark was created for the purpose of comparing formats across multiple programming languages, features and interfaces using standard metrics to evaluate real world performance as accurately as possible. This benchmark was deployed in highly specialized cluster environments such as Levante, which are purpose-built for earth data analysis.

Features, such as chunked and contiguous layouts, were explored in detail, as well as newer offerings like subfiling and asynchronous I/O that are currently only available in selected formats. Parallelism, as an important factor, was evaluated wherever equivalent options were available, as these types of workloads represent the usual tasks clusters, like Levante, are designed for.

6.1. Future Work

Profiling the cluster for anomalies

As shown in Section 5.2, the cluster currently does seem to suffer from some anomalies, affecting some formats like HDF5 and NetCDF4. They have the potential to seriously impact performance and therefore waste precious resources. Such anomalies should be analyzed carefully, as finding the root cause on complex compute clusters is a non-trivial task. Tools, such as Darshan, might be helpful to profile the benchmark at run-time and narrow down potential causes. They can also offer greater insight to the unique behaviors displayed by each format, allowing for more fine grained analysis of potential bottlenecks and shortcomings such as memory footprint.

Extending the benchmark framework

The benchmark framework features a simple *read-to-memory* task in its current iteration. However this does not cover all use-cases. Testing different configurations with compression, independent I/O, Zarr with MPI, writing datasets, higher number of ranks, more nodes or even trying to port C specific features to Python, will help to understand these formats and their limitations better. Evaluating chunking for other file sizes and conceptualizing tasks that can take advantage of chunked datasets, a lot is left to explore in order to carve out the spaces where Zarr, NetCDF4 and HDF5 can be the most helpful. The benchmark is written specifically to be easily extendable, to allow for new features and configurations to be added in the future.

Bibliography

[Adobe, 2025] Adobe, ©. 2025. (2025). MKV files. Retrieved from https://www.adobe.com/creativecloud/file-types/video/container/mkv.html# (accessed May 23, 2025)

[Ambatipudi and Byna, 2023] Ambatipudi, S., and Byna, S. (2023). A Comparison of HDF5, Zarr, and netCDF4 in Performing Common I/O Operations. Retrieved from https://arxiv.org/abs/2207.09503

[Bartz et al., 2015] Bartz, C., Chasapis, K., Kuhn, M., Nerge, P., and Ludwig, T. (2015). A Best Practice Analysis of HDF and NetCDF- Using Lustre. Retrieved from https://doi.org/10.1007/978-3-319-20119-1_20 (accessed May 23, 2025)

[Buettner et al., 2009] Buettner, D., Kunkel, J., and Ludwig, T. (2009). Using Non-blocking I/O Operations in High Performance Computing to Reduce Execution Times — doi.org. Retrieved from https://doi.org/10.1007/978-3-642-03770-2_20

[Copeland, 2006] Copeland, B. J. (2006). The Modern History of Computing. Retrieved from https://plato.stanford.edu/entries/computing-history/ (accessed May 23, 2025)

[Farrell, 2023] Farrell, P. (2023). LAD 2023: Buffered I/O, DIO & Unaligned DIO. Retrieved from https://www.eofs.eu/wp-content/uploads/2024/02/04-LAD-2023-Unaligned-DIO.pdf (accessed May 23, 2025)

[Folk et al., 2011] Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. Uppsala, Sweden: Association for Computing Machinery.

[Godoy et al., 2025] Godoy, J., Hogbin, E. J., Komarinski, M. F., and Merrill, D. C. (2025). Linux System Administrators Guide: Chapter 5. Using Disks and Other Storage Media. Retrieved from https://tldp.org/LDP/sag/html/filesystems.html (accessed May 23, 2025)

[The HDF Group, 2006] The HDF Group. (2006). Selecting hyperslabs. Retrieved from https://docs.hdfgroup.org/archive/support/HDF5/doc/H5.intro.html (accessed May 23, 2025)

[HDFGroup, 2009] HDFGroup. (2009). HDF5 FAQ – How is HDF5 different than HDF4 ?. Retrieved from https://web.archive.org/web/20090330052722/http://www.hdfgroup.org/h5h4-diff.html (accessed May 23, 2025)

[HDFGroup, 2023] HDFGroup. (2023). A Brief History of HDF5 - Mike Folk, one of the original founders of the Hierarchical Data Format. Retrieved from https://www.youtube.com/watch?v=fYYXuq85v0U&t=4s (accessed May 23, 2025)

[HDFGroup, 2025c] HDFGroup. (2025c). Chunking in HDF5. Retrieved from https://support.hdfgroup.org/documentation/hdf5/latest/hdf5_chunking.html (accessed May 23, 2025)

[HDFGroup, 2025b] HDFGroup. (2025b). HDF5 Virtual Object Layer (VOL). Retrieved from https://support.hdfgroup.org/documentation/hdf5/latest/_h5_v_l__u_g.html (accessed May 23, 2025)

[HDFGroup, 2025a] HDFGroup. (2025a). The HDF5 Data Model and File Structure. Retrieved from https://support.hdfgroup.org/documentation/hdf5/latest/_h5_d_m__u_g.html (accessed May 23, 2025)

[henderson, 2023] henderson, J. (2023). HDF5 Subfiling VFD User's Guide. Retrieved from https://github.com/HDFGroup/hdf5doc/blob/master/RFCs/HDF5_Library/VFD_Subfiling/user_guide/HDF5_Subfiling_VFD_User_s_Guide.pdf (accessed May 23, 2025)

[Henderson and Breitenfeld, 2021] Henderson, J., and Breitenfeld, S. (2021). HDF5 VFD Plugins. Retrieved from https://www.hdfgroup.org/wp-content/uploads/2021/10/HDF5-VFD-Plugins-HUG.pdf (accessed May 23, 2025)

[Henderson and Breitenfeld, 2022] Henderson, J., and Breitenfeld, S. (2022). HDF5 Subfiling VFD. Retrieved from https://www.hdfgroup.org/wp-content/uploads/2022/09/HDF5-Subfiling-VFD.pdf (accessed May 23, 2025)

[Jones, 2006] Jones, M. (2006). Boost application performance using asynchronous I/O. Retrieved from https://developer.ibm.com/articles/l-async/ (accessed May 23, 2025)

[Mellanox-Technologies-Inc, 2024] Mellanox-Technologies-Inc. (2024). Introduction to InfiniBand. Retrieved from https://network.nvidia.com/pdf/whitepapers/IB_Intro_WP_190.pdf (accessed May 23, 2025)

[Miles, 2019] Miles, A. (2019). Zarr: Scalable Storage of Tensor Data for Use in Parallel and Distributed Computing | SciPy 2019. Retrieved from https://www.youtube.com/watch?v=qyJXBlrdzBs&list=PLvkeNUPrCU04Xvcph4ErxsRkZq28Oucr7 (accessed May 23, 2025)

[Miles et al., 2024] Miles, A., Striebel, J., Rzepka, N., Maitin-Shepard, J., and Moore, J. (2024). Zarr specs documentation Version 1. Retrieved from https://zarr-specs.readthedocs.io/en/latest/v1/v1.0.html (accessed May 23, 2025)

[Miles et al., 2025c] Miles, A., Striebel, J., Rzepka, N., Maitin-Shepard, J., and Moore, J. (2025c). Parallel computing and synchronization. Retrieved from https://zarr.readthedocs.io/en/stable/user-guide/performance.html#parallel-computing-and-synchronization (accessed May 23, 2025)

[Miles et al., 2025b] Miles, A., Striebel, J., Rzepka, N., Maitin-Shepard, J., and Moore, J. (2025b). Terminology Hierarchy. Retrieved from https://zarr-specs.readthedocs.io/en/latest/_images/terminology-hierarchy.excalidraw.png (accessed May 23, 2025)

[Miles et al., 2025a] Miles, A., Striebel, J., Rzepka, N., Maitin-Shepard, J., and Moore, J. (2025a). Zarr. Retrieved from https://zarr-specs.readthedocs.io/en/latest/v3/core/index.html#store (accessed May 23, 2025)

[NumPy Developers, 2025] NumPy Developers. (2025). Indexing on ndarrays. Retrieved from https://numpy.org/doc/stable/user/basics.indexing.html (accessed May 23, 2025)

[NumPy Developers, 2024] NumPy Developers, ©. Copyright 2008-2024. (2024). numpy.mean. Retrieved from https://numpy.org/doc/stable/reference/generated/numpy.mean.html (accessed May 23, 2025)

[OpenSFS and EOFS, 2017] OpenSFS, ©. 2025, and EOFS. (2017). Introduction to Lustre Architecture. Retrieved from https://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf (accessed May 23, 2025)

[OpenSFS and EOFS, 2025] OpenSFS, ©. 2025, and EOFS. (2025). Progressive File Layouts. Retrieved from https://wiki.lustre.org/Progressive_File_Layouts (accessed May 23, 2025)

[Regents, 2023] Regents, ©. 2. N. M. S. U. - Board of. (2023). Message Passing Interface. Retrieved from https://hpc.nmsu.edu/discovery/mpi/introduction/ (accessed May 23, 2025)

[Rew and Davis, 1990] Rew, R., and Davis, G. (1990). NetCDF: an interface for scientific data access. *IEEE Computer Graphics and Applications*, *10*(4), 76–82.

[Rew et al., 2025b] Rew, R., Davis, G., Emmerson, S., Davies, H., Hartnett, E., Heimbigner, D., and Fisher, W. (2025b). How can I convert HDF5 files into netCDF-4 files?. Retrieved from https://docs.unidata.ucar.edu/netcdf-c/current/faq.html#How-can-I-convert-HDF5-files-into-netCDF-4-files (accessed May 23, 2025)

[Rew et al., 2025a] Rew, R., Davis, G., Emmerson, S., Davies, H., Hartnett, E., Heimbigner, D., and Fisher, W. (2025a). The NetCDF Data Model. Retrieved from https://docs.unidata.ucar.edu/netcdf-c/current/netcdf_data_model.html (accessed May 23, 2025)

[Susnjara and Smalley, 2024] Susnjara, S., and Smalley, I. (2024). What is high-performance computing (HPC)?. Retrieved from https://www.ibm.com/think/topics/hpc (accessed May 23, 2025)

[The HDF Group and Illinois, 2017] The HDF Group, ©., and Illinois. (2017). HDF Specification and Developer's Guide. Retrieved from https://support.hdfgroup.org/archive/support/release4/doc/DS.pdf (accessed May 23, 2025)

[The Regents of the University of California, 2020] The Regents of the University of California, ©. Copyright Copyright (c) 2020. (2020). HDF5 Asynchronous I/O VOL Connector. Retrieved from https://hdf5-vol-async.readthedocs.io/en/latest/ (accessed May 23, 2025)

[Unidata, 2018b] Unidata, ©. 2018. (2018b). How-many-netCDF-formats-are-there-and-what-are-the-differences-among-them. Retrieved from https://docs.unidata.ucar.edu/netcdf-c/current/faq.html#How-many-netCDF-formats-are-there-and-what-are-the-differences-among-them (accessed May 23, 2025)

[Unidata, 2018a] Unidata, ©. 2018. (2018a). NetCDF: FAQ. Retrieved from https://docs.unidata.ucar.edu/netcdf-c/current/faq.html (accessed May 23, 2025)

[Zarr, 2022] Zarr, ©. 2024. (2022). Zarr. Retrieved from https://zarr.dev/ (accessed May 23, 2025)

[Zarr, 2024] Zarr, ©. 2024. (2024). Zarr Releases. Retrieved from https://github.com/zarr-developers/zarr-python/releases?page=1 (accessed May 23, 2025)

[© Deutsches Klimarechenzentrum GmbH, 2025] © Deutsches Klimarechenzentrum GmbH. (2025). Levante - a new Supercomputer for Earth System Research. Retrieved from https://www.dkrz.de/en/projects-and-partners/projects/focus/levante-spotlight (accessed May 23, 2025)

Appendix A.

Appendix

```
==spack packages==

gcc             11.2.0
openmpi         5.0.5
hdf5            1.14.5
netcdf-c        4.9.2
hdf5-vol-async  develop
argobots        main
python          3.11.9
py-pip          23.1.2
py-mpi4py       4.0.1
py-netcdf4      1.7.1
py-h5py         3.12.1


==pip packages==

zarr            3.0.5
gprof2dot       2024.6.6
xarray          2025.1.2
matplotlib      3.10.1
pandas          2.2.3
schedule        1.2.2
plotly          6.0.0
kaleido         0.2.1
nbclean         0.3.2
```

Listing A.1.: Software Stack

```
#!/bin/bash
#SBATCH --wait
#SBATCH --partition=compute
#SBATCH --account=ku0598
#SBATCH --constraint="[cell02]"
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=1
#SBATCH --mem=0
#SBATCH --cpu-freq=High
#SBATCH --distribution=block:cyclic
#SBATCH --time=08:00:00
#SBATCH --output=log/log-%j/log.%j.txt
#SBATCH --error=log/log-%j/log.%j.err


# Begin of section with executable commands
set -e
ls -l

$1
```

Listing A.2.: Example sbatch script

```
# general logic:
#
# check if in compute partition - then the nodes were used exclusively and we don't need to pay attention to other jobs/users
#
# if not in compute partition then
#    check if you have yourself another job on the node
#              --> if yes, don't do anything and exit
#
#    check if there are other jobs on the node
#          --> if yes, delete your processes and delete your files from /dev/shm
#          --> if no, in addition change governors etc

flap_report(){}

cleanup_user_processes() {
    # Don't try to kill user root or system daemon jobs

    # No other SLURM jobs from this user, purge all remaining processes of this user
}

cleanup_temp_for_user() {}

cleanup_temp_complete() {  }

complete_clean() {
    cleanup_temp_complete
    cleanup_user_processes
    transit_to_idle
}

dkrz_epilog_report() {}

######################################################################
# MAIN
######################################################################

# DKRZ reporting on single node only

    # for now we do not report on hetjobs ...

    # jobs of this user are running

    # jobs are running, but not from this user, otherwise would have excited

# no jobs are running
```

Listing A.3.: slurm epilog script

```
--constraint="[cell02]"
    l10450
    l10454
    l10480
    l10500
    l10579
```

Listing A.4.: Software Stack

Statement of Authorship

Thesis Title:
Analyzing I/O Efficiency in Zarr, HDF5, and NetCDF4

Name: Lucas Philipp

Date of Birth:

Matriculation Number:

I herewith assure that I wrote the present thesis independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Furthermore, I confirm that I am aware that the use of content (including but not limited to text, figures, images and code) generated by artificial intelligence (AI) in the thesis must be disclosed. In those cases I have specified the AI system used, I have marked the specific sections of the thesis where AI-generated content was used, and I have provided a brief explanation of the level of detail at which the AI system was used to generate the content. I also stated the reason for using the tools.

I assure that even if a generative AI system has been used, the scientific contribution has been made entirely by myself.

Magdeburg, May 23, 2025

_____

Signature