A Comparison of CSV, HDF5, Zarr, and netCDF4 in Performing Common I/O Operations

Sriniket Ambatipudi
Dulles High School
Houston, TX, USA
Email: ambatipudisriniket@gmail.com

Yusen Peng
The Ohio State University
Columbus, OH, USA
Email: yusenrocks@gmail.com

Suren Byna
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
Email: sbyna@lbl.gov

***Abstract*—Scientific data is often stored in files because of the simplicity they provide in managing, transferring, and sharing data. These files are typically structured in a specific arrangement and contain metadata to understand the structure the data is stored in. There are numerous file formats in use in various scientific domains that provide abstractions for storing and retrieving data. With the abundance of file formats aiming to store large amounts of scientific data quickly and easily, a question that arises is, "Which scientific file format is best suited for a general use case?" In this study, we compiled a set of benchmarks for common file operations, i.e., create, open, read, write, and close, and used the results of these benchmarks to compare three popular formats: HDF5, netCDF4, and Zarr.**

GENERAL TERMS / KEYWORDS

Scientific File Formats, HDF5, netCDF4, Zarr

I. INTRODUCTION

With the rapid advancement of experiments, observations, and simulations in recent years, various domains of science are producing enormous amounts of data. For example, the Large Hadron Collider (LHC) experiments at CERN produce 90 petabytes of data per year [1]. NASA's Climate Data Services (CDS) simulate our planet's weather and climate models from hours to millennia and produce datasets up to petabytes in volume [2].

Much of the data produced in scientific experiments, observations, and simulations are stored in files with various formats. Scientific file formats offer a medium to store scientific data for long-term processing, which is of great importance to researchers. Each file format has specific characteristics that make it suited for a particular use case, and this makes choosing the appropriate file format an important task. The typical content of scientific files includes data with a structure and metadata describing the structure and the data within. Data in these files are often structured as arrays [3]. The metadata that describes the data often contains the origins of the data, configurations used in generating or collecting the data, and the location of the data in the file format for easy access. As such, scientific file formats will often provide functionality for the storing and retrieval of data and metadata in files.

There are numerous file formats in existence, such as HDF5 [4], netCDF4 [5], ROOT [6], Zarr [7], and many more [8]. Each of these file formats exhibits different performance characteristics and was designed to accomplish a specific task. For example, the High Energy Physics (HEP) community developed the ROOT framework to meet the high-performance requirements for multithreaded read and write operations and support object-oriented programming. As a result of these design specifications, the ROOT framework is used by CERN in its research with the Large Hadron Collider [9]. On the other hand, file formats such as netCDF4 or HDF5 are often used in more general use case scenarios because of their self-describing capabilities, which allow the storing of metadata to describe the data within a file. Such self-describing capabilities allow these file formats to be used in a multitude of applications, like how HDF5 is used in astronomy, medicine, physics, and many more fields [4].

Because there are a multitude of file formats that are available to store scientific data, the question of which file format is best suited for a general use case arises. Previous research [10, 11] has mainly focused on testing the performance characteristics of individual file formats, like how HDF5 was tested for its performance in reading a subset of a large array [10] or how netCDF4's performance characteristics were analyzed [11]. The first test revealed that when working with an HDF5 file in Python, the fastest way to read data is to memory map the file with NumPy, bypassing the HDF5 Python API (h5py). Memory mapping involves mapping a file's contents into memory, and this means that data within a file can be accessed if the location of the data in terms of an offset is known. The hierarchical structure of file formats such as HDF5 means that accessing the data within is relatively simple, as the file's metadata stores the location of individual datasets. This test was useful in analyzing the shortcomings of HDF5 in a particular use case, allowing potential users to reconsider whether the HDF5 file format would be best suited for their use case. In the second experiment, conducted by the HDF Group, the netCDF4 file format was tested for its performance characteristics and then compared to its predecessor, netCDF3. The results of this experiment showed that netCDF4 generally had slower write speeds than netCDF3, but it had faster read speeds due to netCDF4's use of the HDF5 library internally.

This type of testing is useful for analyzing the performance characteristics of one file format and its shortcomings in specific use cases, but when the performance characteristics of one file format must be compared to the performance characteristics of other file formats, a benchmark offers

itself as a viable option because it allows for the objective measurement of the speed at which each file format is able to perform a specific task. Such a benchmark has the potential to be a valuable asset to researchers, as it allows them to choose the file format that is not only suited for their particular use case but also performs the best in comparison to its alternatives. This allows the researchers to have ease of access when storing and modifying data at fast speeds, allowing more time and effort to be put elsewhere in their project. In this work, we developed a benchmark to compare the read and write speeds of three multipurpose scientific file formats (HDF5, netCDF4, and Zarr). This benchmark writes randomized data to a specified number of datasets within a file and measures the time taken to write the data to each dataset and the time taken to read the contents of each dataset, allowing objective comparisons to be drawn between the three file formats' performance in different operations.

In the remainder of the paper, we first provide a brief background in §II to the three file formats we used in our evaluation in this study. In §III, we describe the read and write benchmarks we used in the evaluation. In §IV, we provide details of the system we used for comparing and evaluating the performance of the three file formats under different workloads.

II. Background

A. HDF5

HDF5, or the Hierarchical Data Format 5, is a file format designed to store a large amount of data in an organized manner. Typically characterized by a `.hdf5` or `.h5` file extension, this file format stores data in a manner very similar to that of a file system. This file format's primary data models are groups and datasets. Groups are the overarching structure, and they can hold other groups or datasets. Datasets store raw data values of a specified data type and are usually stored within groups [12]. A feature of HDF5 is that it is able to store data consisting of different data types within the same file [13]. As mentioned earlier, this file format is self-describing, meaning that all the groups and datasets within the file format contain metadata describing their contents. This allows for the data within the file to be mapped in memory, provided the API supports it. Generally speaking, users use the HDF5 API to issue commands to a lower-level driver, which is in charge of accessing the file and performing the requested operations [14]. Because the file format is open source, there has been widespread API support across most modern languages (Python, C++, and Java).

B. netCDF4

netCDF4 is a file format that is designed to store array-oriented data and is characterized by a `.netc` file extension. It stores data in a manner similar to HDF5, with groups serving as the overarching data structure. Within a group, there can be other groups or variables. Variables are akin to HDF5 datasets. Unlike HDF5 datasets, netCDF4 variables cannot be resized once they are created [15]. To circumvent this, variables can be declared with an unlimited size in a specified dimension. Similar to HDF5, netCDF4 is a self-describing file format, and this means that groups and variables both contain metadata describing their contents. Unlike its predecessor, netCDF3, netCDF4 uses HDF5 as its backend, allowing it to achieve faster read times [11].

C. Zarr

Zarr is a file format that is designed to store large arrays of data and is characterized by a `.zarr` file extension. Because it is based on NumPy, it is geared mainly towards Python users. Similar to both HDF5 and netCDF4, Zarr is also a hierarchical, self-describing file format that has groups as the overarching file structure. Each group contains datasets, which are representative of multidimensional arrays of a homogeneous data type. Furthermore, the API for this file format was designed to be similar to h5py (HDF5's Python API), and as a result, it includes functions based on h5py's functions, namely the group creation function [16]. One advantage to using Zarr is that it provides multiple options to store data by allowing a user to store a file in memory, in the file system, or in other storage systems with a similar interface to the first two options [16].

III. Benchmarks

As a benchmark is being used to compare the performance of the file formats, the benchmark must only test features of the file format that are supported by all the file formats being tested. This benchmark compares the time taken to create a dataset, write data to a dataset, and finally open that dataset at a later time and read its contents. This can be categorized into two main types of operations—the writing operation and the reading operation. Both are very important features to test in a file format, as the end goal of a file format is to store data for long-term processing. The faster write and read times are not only indicative of better performance characteristics but also have a tangible effect on the improvement of an end-user's workflow, as less time would be spent performing operations that are not directly relevant to the task at hand.

To allow for greater flexibility when benchmarking the file formats, we added a configuration system in which the user is able to specify the testing parameters such as the number of datasets to create within the file and the dimensions of the array that will be written to each dataset by editing a `.yaml` configuration file. After the benchmark is done, the program then stores the times taken across multiple trials in a `.csv` file and plots the data in the `.csv` file with `matplotlib.pyplot` to allow a user to make a definitive comparison between the file formats being tested. Below, the main operations, the write operation and the read operation, will be discussed in-depth.

A. Write Benchmark

The write operation is the first operation to be tested in the benchmark. It creates files with the filename as specified in the configuration file and extensions `.hdf5` for HDF5 files, `.netc` for netCDF4 files, and `.zarr` for Zarr files. The file is placed inside a folder named `Files/`, to help

reduce clutter in the working directory. Taking information from the configuration file, a sample data array is generated with dimensions and length as specified. This sample data array consists of randomly-generated 32-bit floats. Then, the program creates a dataset within the file and writes the sample data array to the dataset. This process of generating a sample data array, creating a dataset, and populating it with the values from the sample data array is repeated until the benchmark has created the number of datasets as specified by the configuration file. After the file is populated with data, the benchmark copies the file to a directory named `Files Read/` and renames the file to avoid any caching effects that may interfere with the read times. There are numerous ways to mitigate such caching effects, such as waiting for an extended period of time, but simply moving the file to another directory and renaming the file is the quickest and easiest way to mitigate the effects of caching in interfering with the times taken to read from the file. The time taken to create all the datasets and populate them with data is divided by the number of datasets to find the average time taken to create and populate one dataset. Both of these times are then returned to the main program, where they are written to the `.csv` output file.

In theory, netCDF4 generally writes faster if all datasets are created before being written, compared to the case where each dataset is written immediately after being created. Thereby, the loop structure is changed from single loop (each dataset is written immediately after being created) to separate loop (all datasets are created before being written). In the experiment, two plots are made to validate the optimization above.

The I/O performance of `csv` file format is also measured and compared to HDF5, netCDF4, and zarr. Since creating and opening dataset inside `csv` file is not applicable, only write and read benchmarks are developed and tested. Besides, since `csv` files are not typically used in the context of tensors, only vectors and matrices are used for the dimension of random data. As for the write benchmark, `NumPy` library in Python is utilized. An empty dataset is created by `np.empty` function and one `csv` file is created by `np.savetxt` to save the dataset as a `csv` file. When it comes to write-time measurement, `np.loadtxt` is used first to load the `csv` file as a dataset. The dataset is populated with random data and saved as `csv` file again by calling `np.savetxt` function. Note that `np.loadtxt` and `np.savetxt` are applicable for both vectors and matrices without explicitly reshaping dataset arrays. All individual `csv` files are stored in the `CSV data` directory.

In this benchmark project, the effect of compression on file common I/O operation performance is investigated. Specifically, blosc compression is used to compress HDF5, netCDF4, and Zarr files and the performance of compressed file formats is measured and compared with uncompressed versions of corresponding file formats. In terms of implementation details, compressed HDF5, compressed netCDF4, compressed zarr are treated as different individual data models to facilitate the process of developing code. A total of 6 `csv` files are created to keep track of common I/O operation performance to prepare for plotting. `matplotlib.pyplot` is utilized to visualize file I/O operation performance of HDF5, netCDF4, Zarr, and corresponding compressed versions of them on a single bar plot.

The compound datatype in HDF5 is a similar data model to `csv` files. In this benchmark project, the performance of write/read operations are measured for both HDF5 compound datatype and csv files. Specifically, in the write benchmark for HDF5 compound datatype, the random data are written into one single compound dataset by properties; However, in the case of `csv` files, random data could be written either by columns or by rows. Accordingly, both writing approaches are considered and implemented respectively. To write random data by columns, the `dataframe` in `pandas` is utilized to populate data into columns. In order to write data by rows, on the other hand, the `csv.writer` in `csv` module is used to write data in a row-wise fashion.

B. Read Benchmark

The benchmark now opens the copied file in the `Files Read/` directory and begins testing the read operations of the three file formats. This operation consists of opening each dataset within the file and printing its contents to the standard output. The time taken to open all the datasets and the time taken to read from all the datasets are once again divided by the number of datasets within the file to find the average time taken to open and read one dataset. Both of these times are then returned to the main program, where they are also written to the `.csv` output file. In terms of reading `csv` files, `np.loadtxt` function is used. In the read benchmark for compound datatype, four different reading approaches are proposed and implemented respectively: read data by columns; read data by rows; read the entire dataset; read data by the first half of rows. All reading approaches are implemented by `dataframe` and `df.iloc` function in `pandas` library.

This process of running the write operation benchmark and the read operation benchmark is then repeated multiple times in order to ensure the consistency of the data gathered. To avoid filling up the disk with generated test files, the `Files/` and `Files Read/` directories are deleted between trials. Finally, the data from the `.csv` file are averaged out with `pandas` and plotted with `matplotlib.pyplot` to allow for visualizing a comparison between the tested file formats in a given operation.

C. Elimination of Caching Effect

The ”write all, read all” approach in the `runner.py` python file is used to eliminate any caching effect. In other words, ”write all, read all” means to finish writing the random data into datasets associated with all file formats (HDF5, netCDF4, zarr) before reading them. This helps avoid any caching effect when reading files. In terms of implementation details, `dictionary` and `list` in Python are applied. Specifically, one dictionary is created for every single I/O operation: create, write, open, read. These dictionaries are

mapping each file format to its performance data list of the corresponding I/O operation. These data lists keep track of the performance measurements from all of repeated trials. In this experiment, the number of repeated trials is 5. Dictionaries for create, write, open, read benchmarks are utilized to store and transfer performance data to `csv` files later for plotting.

IV. Performance Evaluation

A. Experimental setup

The three file formats were tested on a computer running Ubuntu 18.04.5 with an Intel(R) Xeon(R) Silver 4215R CPU, 196 Gigabytes of RAM, and 960 Gigabytes of solid-state storage provided by a Micron 5200 Series SSD. The version of `h5py` used to test the HDF5 file format was `3.6.0`. The version of `netCDF4` used to test the netCDF4 file format was `1.5.8`. The version of `zarr` used to test the Zarr file format was `2.11.0`.

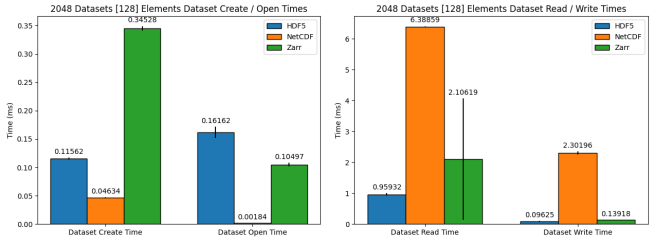
The benchmark parameters that were used in each run of the test can be found in the tables to the right. Note that the `Test Name` parameter is automatically generated by the benchmark and is used to create the generated plot's title.

In order to identify the performance trend or pattern with different scales of increasing number of elements, the scale element comparison section is added to this benchmark project. The number of elements is increasing in four different scales, whereas the number of datasets is fixed. The scale element comparison is applied to both basic comparison (comparison among HDF5, netCDF4, and zarr) and compression comparison (HDF5, netCDF4, zarr, and corresponding compressed versions). In terms of the implementaion details, various data structures like `dictionary` and `list` in Python are applied. Basically, one single dictionary called `database` is created to map each individual file format to its data list, which stores the `average time` and `standard deviation` of create, write, open, read performance. When it comes to `matplotlib.pyplot` plotting process, the performance data including `average time` and `standard deviation` will be retrieved by its file format and eventually visualize the performance comparison.

B. Data

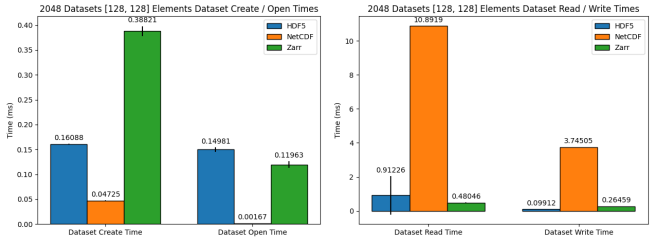
1) Small-Scale Testing: small-scale testing:

ID	Number of Variables	Array Dimensions	Data Size in MB
Small Scale Testing			
1	2048	[128]	1
2	2048	[128, 128]	128
3	2048	[128, 128, 128]	16384
4	2048	[256]	2
5	4096	[256]	4
6	8192	[256]	8
netCDF4 Optimization			
7	2048	[1048576]	8192
Comparison with CSV			
8a	2048	[1048576]	8192
8b	2048	[1024, 1024]	8192
Large Scale Testing - Basic Comparison			
9	2048	[1048576]	8192
10	2048	[1024, 1024]	8192
11	2048	[128, 128, 64]	8192
12	4096	[1048576]	16384
13	4096	[1024, 1024]	16384
14	4096	[128, 128, 64]	16384
15	64	[1073741824]	262144
16	64	[1048576, 1024]	262144
17	64	[1024, 1024, 1024]	262144
18	128	[1073741824]	524288
19	128	[1048576, 1024]	524288
20	128	[1024, 1024, 1024]	524288
Scale Element Comparison - Basic Comparison			
21	64	[1048576],[10485760], [104857600], [1073741824]	256, 2560, 25600, 262144
Large Scale Testing - Compression Comparison			
22	2048	[1048576]	8192
23	2048	[1024, 1024]	8192
24	4096	[1048576]	16384
25	4096	[1024, 1024]	16384
26	4	[10485760]	160
27	4	[104857600]	1600
Scale Element Comparison - Compression Comparison			
28	4 in HDF5	[1048576],[10485760], [104857600], [1073741824]	256, 2560, 25600, 262144
29	4 in netCDF4	[1048576],[10485760], [104857600], [1073741824]	256, 2560, 25600, 262144
30	4 in Zarr	[1048576],[10485760], [104857600], [1073741824]	256, 2560, 25600, 262144
Compound Datatype Comparison			
31	512	[1024]	2
32	1024	[1024]	4
33	4	[1048576]	16



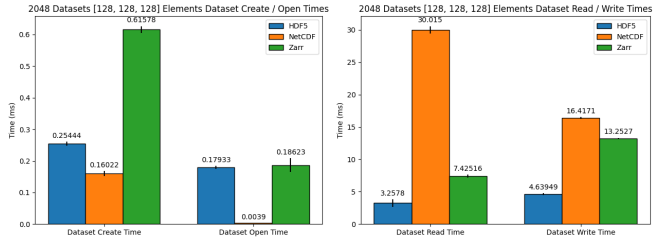
(a) Create / Open Times (b) Read / Write Times

Fig. 1: Small scale testing: 2048 Datasets of [128] Elements; average create time, average open time, average read time, and average write time per dataset



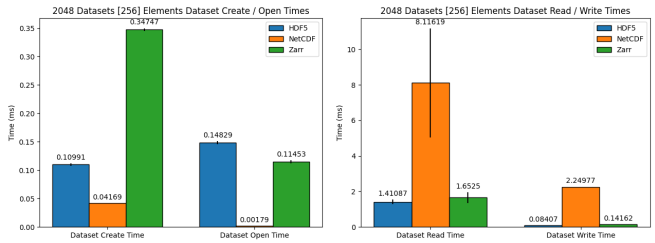
(a) Create / Open Times (b) Read / Write Times

Fig. 2: Small scale testing: 2048 Datasets of [128, 128] Elements; average create time, average open time, average read time, and average write time per dataset



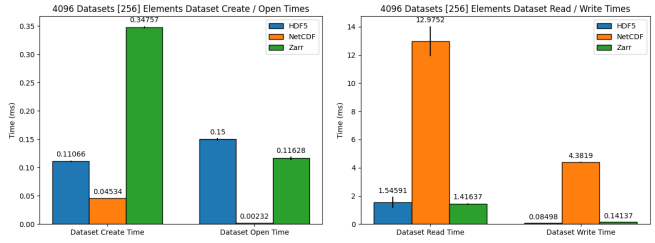
(a) Create / Open Times (b) Read / Write Times

Fig. 3: Small scale testing: 2048 Datasets of [128, 128, 128] Elements; average create time, average open time, average read time, and average write time per dataset



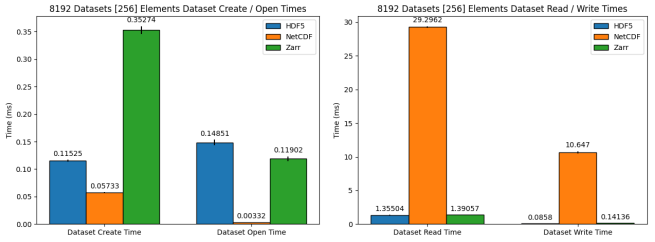
(a) Create / Open Times (b) Read / Write Times

Fig. 4: Small scale testing: 2048 Datasets of [256] Elements; average create time, average open time, average read time, and average write time per dataset



(a) Create / Open Times (b) Read / Write Times

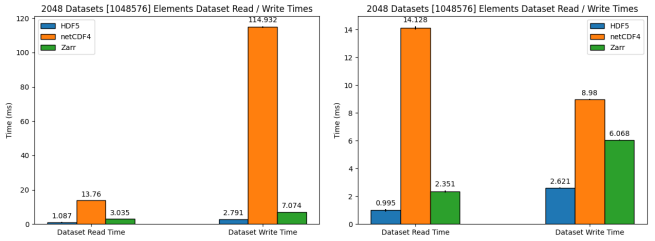
Fig. 5: Small scale testing: 4096 Datasets of [256] Elements; average create time, average open time, average read time, and average write time per dataset



(a) Create / Open Times (b) Read / Write Times

Fig. 6: Small scale testing: 8192 Datasets of [256] Elements; average create time, average open time, average read time, and average write time per dataset

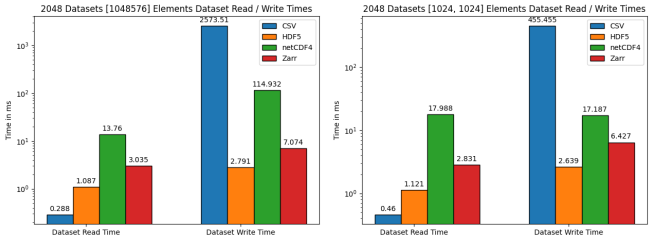
2) netCDF4 Optimization: netCDF4 Optimization:



(a) Combined Loop Write Times (b) Separate Loop Write Times

Fig. 7: netCDF4 Optimization: a comparison of combined loop versus separate loop with 2048 Datasets of [1048576] Elements; average create time, average open time, average read time, and average write time per dataset

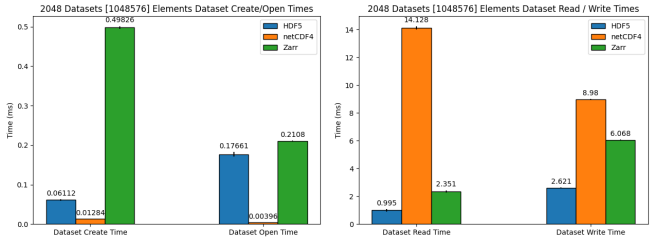
3) Comparison with CSV: Comparison with CSV



(a) Read / Write Times (b) Read / Write Times

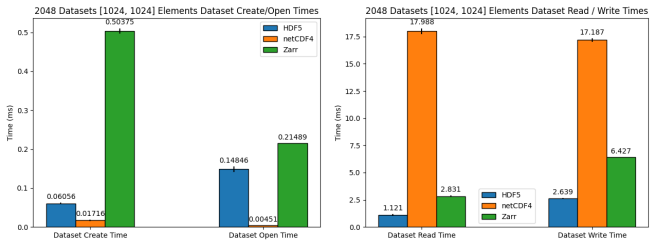
Fig. 8: Comparison with CSV in read/write time with 2048 Datasets of 1048576 Elements; average create time, average open time, average read time, and average write time per dataset

4) Large-Scale Testing - Basic Comparison: Large-Scale Testing



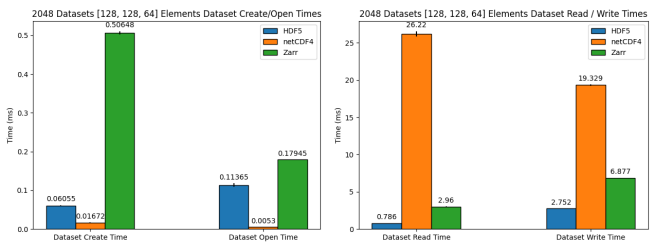
(a) Create / Open Times (b) Read / Write Times

Fig. 9: Large Scale Testing: 2048 Datasets of [1048576] Elements; average create time, average open time, average read time, and average write time per dataset



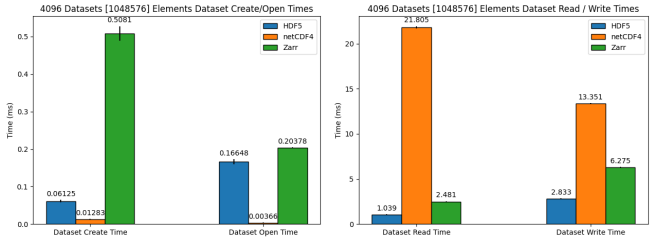
(a) Create / Open Times (b) Read / Write Times

Fig. 10: Large Scale Testing: 2048 Datasets of [1024, 1024] Elements; average create time, average open time, average read time, and average write time per dataset



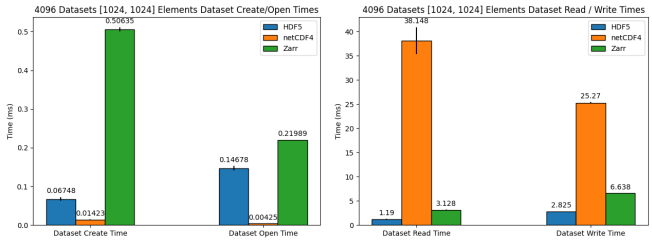
(a) Create / Open Times (b) Read / Write Times

Fig. 11: Large Scale Testing: 2048 Datasets of [128, 128, 64] Elements; average create time, average open time, average read time, and average write time per dataset



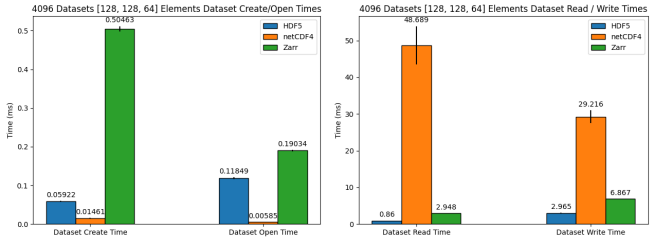
(a) Create / Open Times (b) Read / Write Times

Fig. 12: Large Scale Testing: 4096 Datasets of [1048576] Elements; average create time, average open time, average read time, and average write time per dataset



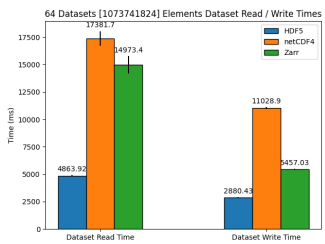
(a) Create / Open Times (b) Read / Write Times

Fig. 13: Large Scale Testing: 4096 Datasets of [1024, 1024] Elements; average create time, average open time, average read time, and average write time per dataset



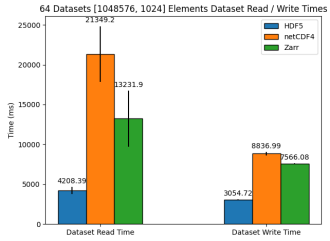
(a) Create / Open Times (b) Read / Write Times

Fig. 14: Large Scale Testing: 4096 Datasets of [128, 128, 64] Elements; average create time, average open time, average read time, and average write time per dataset



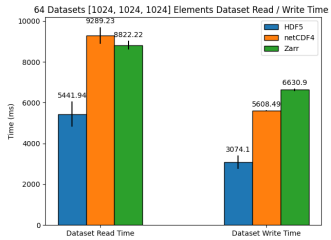
(a) Read / Write Times

Fig. 15: Large Scale Testing: 64 Datasets of [1073741824] Elements; average read time and average write time per dataset



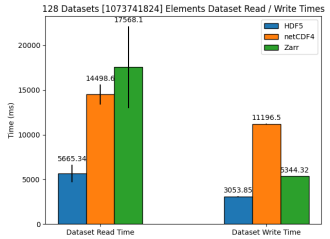
(a) Read / Write Times

Fig. 16: Large Scale Testing: 64 Datasets of [1048576, 1024] Elements; average read time and average write time per dataset



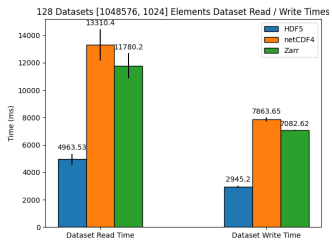
(a) Read / Write Times

Fig. 17: Large Scale Testing: 64 Datasets of [1024, 1024, 1024] Elements; average read time and average write time per dataset



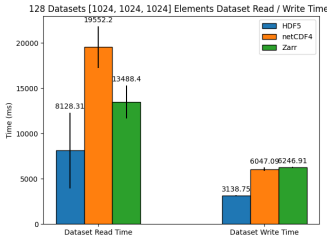
(a) Read / Write Times

Fig. 18: Large Scale Testing: 128 Datasets of [1073741824] Elements; average read time and average write time per dataset



(a) Read / Write Times

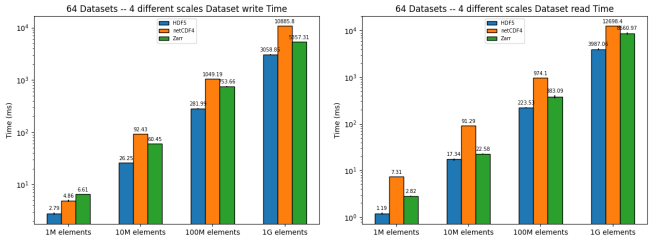
Fig. 19: Large Scale Testing: 128 Datasets of [1048576, 1024] Elements; average read time and average write time per dataset



(a) Read / Write Times

Fig. 20: Large Scale Testing: 128 Datasets of [1024, 1024, 1024] Elements; average read time and average write time per dataset

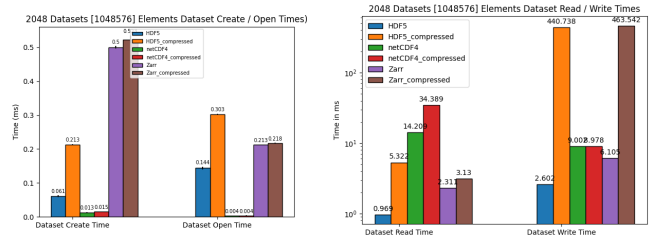
5) Scale Element Comparison - Basic Comparison: Scale Element Comparison



(a) Write Times (b) Read Times

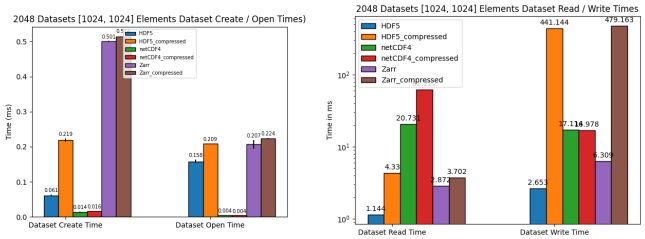
Fig. 21: Scale Element Comparison: 64 Datasets of the number of Elements increasing from 1048576 to 1073741824; average read time and average write time per dataset

6) Large-Scale Testing - Compression Comparison: Large-Scale Testing



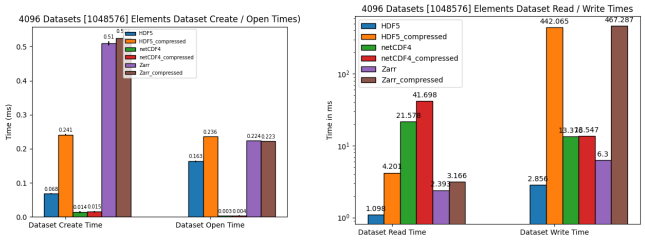
(a) Create / Open Times (b) Read / Write Times log

Fig. 22: Large-Scale Testing - Blosc Compression: 2048 Datasets of [1048576] Elements; average create time, average open time, average read time, and average write time per dataset



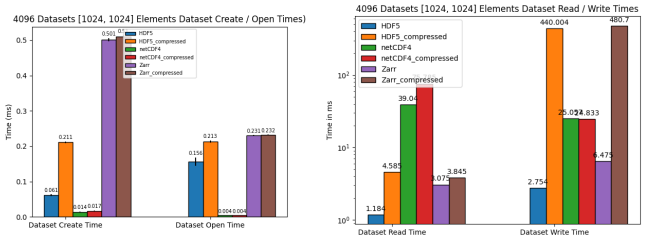
(a) Create / Open Times (b) Read / Write Times log

Fig. 23: Large-Scale Testing - Blosc Compression: 2048 Datasets of [1024, 1024] Elements; average create time, average open time, average read time, and average write time per dataset



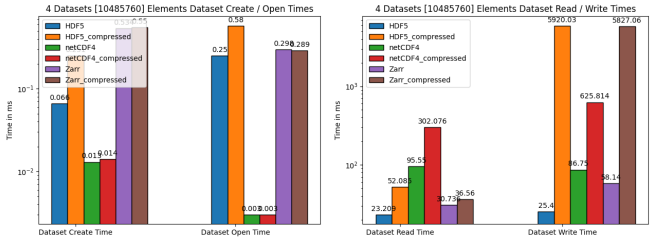
(a) Create / Open Times (b) Read / Write Times log

Fig. 24: Large-Scale Testing - Blosc Compression: 4096 Datasets of [1048576] Elements; average create time, average open time, average read time, and average write time per dataset



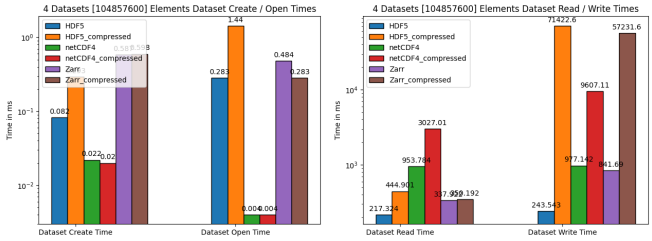
(a) Create / Open Times (b) Read / Write Times log

Fig. 25: Large-Scale Testing - Blosc Compression: 4096 Datasets of [1024, 1024] Elements; average create time, average open time, average read time, and average write time per dataset



(a) Create / Open Times (b) Read / Write Times log

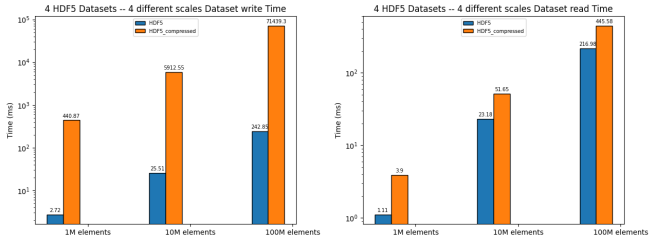
Fig. 26: Large-Scale Testing - Blosc Compression: 4 Datasets of [10485760] Elements; average create time, average open time, average read time, and average write time per dataset



(a) Create / Open Times (b) Read / Write Times log

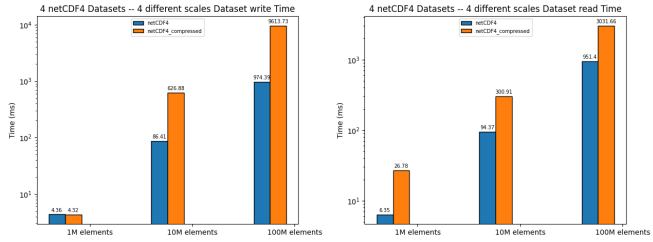
Fig. 27: Large-Scale Testing - Blosc Compression: 4 Datasets of [104857600] Elements; average create time, average open time, average read time, and average write time per dataset

7) Scale Element Comparison - Compression Comparison: Scale Element Comparison



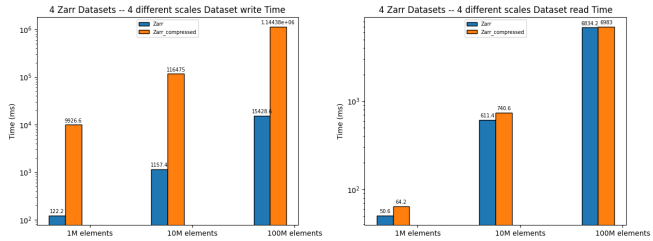
(a) Write Times (b) Read Times

Fig. 28: 4 HDF5 datasets of the number of Elements increasing from 1048576 to 104857600; create time, open time, read time, and write time for total datasets



(a) Write Times (b) Read Times

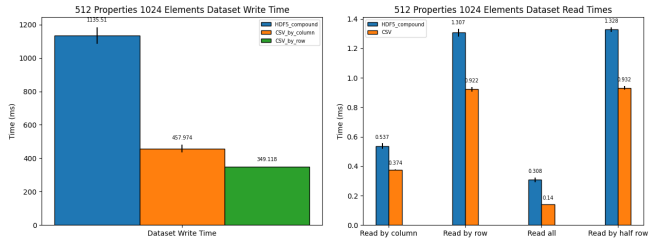
Fig. 29: 4 netCDF4 datasets of the number of Elements increasing from 1048576 to 104857600; create time, open time, read time, and write time for total datasets



(a) Write Times (b) Read Times

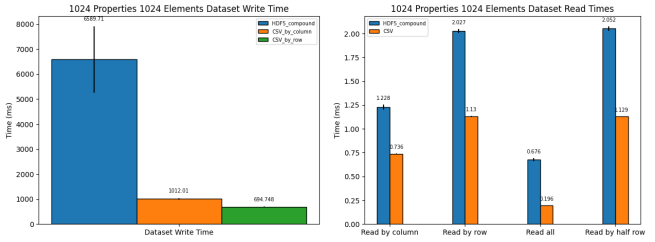
Fig. 30: 4 Zarr datasets of the number of Elements increasing from 1048576 to 104857600; create time, open time, read time, and write time for total datasets

8) Compound Datatype Comparison: Compound Datatype Comparison



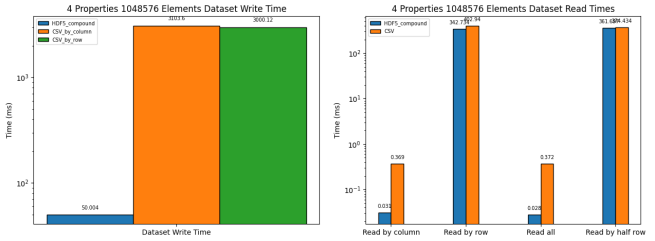
(a) Write Times (b) Read Times

Fig. 31: Compound Datatype Comparison in read/write time: 512 Properties of 1024 Elements; CSV data are written in two different approaches; both HDF5 and CSV are read in four different approaches.



(a) Write Times (b) Read Times

Fig. 32: Compound Datatype Comparison in read/write time: 1024 Properties of 1024 Elements; CSV data are written in two different approaches; both HDF5 and CSV are read in four different approaches.



(a) Write Times (b) Read Times

Fig. 33: Compound Datatype Comparison in read/write time: 4 Properties of 1048576 Elements; CSV data are read in two different approaches; both HDF5 and CSV are read in four different approaches.

C. Discussion

Figure 1 shows the results when 2,048 datasets are created and populated with a one-dimensional array containing 128 32-bit floats. This graph shows that the time taken to create and open a dataset in netCDF4 is much faster than that of HDF5's or Zarr's. In comparison to Zarr, HDF5 takes less time to create a dataset, but it takes slightly more time to open a dataset. When it comes to writing to datasets or reading from datasets, HDF5 and Zarr share very similar times in both operations, with netCDF4 trailing by a large margin.

Figure 2 shows the results when 2,048 datasets are created and populated with a two-dimensional array containing 128 elements in each dimension. The results from this test are almost identical to the results from the previous test, both in terms of the trend and the time taken to complete each operation.

Figure 3 shows the results when 2,048 datasets are created and populated with a three-dimensional array containing 128 elements in each dimension. The results from this test follow the same trend as the past two tests, but the times taken to complete each operation are almost double the times taken in the past two tests.

These past three bar graphs show the tests in which the number of datasets is held constant while increasing the number of dimensions in the data array, but the next three bar

graphs will involve increasing the number of datasets while holding the size of the data array constant in order to measure the effect of increasing the number of datasets on file-format performance.

Figure 4 shows the results when 2,048 datasets are created and populated with a one-dimensional array containing 256 elements. The results from this test mirror those from Figure 1, and this is to be expected as the number of datasets in both tests is the same, with the size of each dataset varying slightly.

Figure 5 shows the results when 4,096 datasets are created and populated with a one-dimensional array containing 256 elements, and Figure 6 shows the results when 8,192 datasets are created and populated with a one-dimensional array containing 256 elements. Both graphs are almost identical to Figure 4, meaning that the number of datasets most likely has no impact on the average time taken to perform the operations requested.

Figure 7 shows how netCDF4 can be optimized in terms of write benchmark by using a different loop structure. The plot (a) illustrates the netCDF4 performance when a combined loop is used; namely, the random data is written into the dataset immediately after the dataset is created. On the contrary, the plot (b) shows netCDF4 performance when a separate loop is used. In other words, the random data will not be written into any dataset until all datasets needed are created in the first place. In comparision with writing data immediately after each dataset is created, netCDF4 performs better in writing data when all tested datasets are created first. It is noted that the separate loop structure has been applied to following experiments as well.

Figure 8 shows how CSV is compared to other file formats in terms of writing and reading random data. Where the plot (a) shows the results when a vector is tested, the plot (b) shows the results when a matrix is tested. It turns out that CSV is generally performing better in reading data, but it performs much worse when it comes to writing random data into datasets. Accordingly, the CSV file format would not be tested in the following experiments.

Figure 9 through 14 shows the performance when the the scale of testing data becomes much larger. The results of figure 9, where the test data is a vector, are consistent with those findings with small-scale data being tested: HDF5 is fastest in reading or writing to a dataset, netCDF4 is fastest in creating or opening a dataset, and Zarr generally trails right behind HDF5 in performance.

Figure 10 shows the performance when the test data is a matrix, with the same data size as Figure 9. Similarly, HDF5 is the best for writing and reading data, whereas netCDF4 is fastest in terms of creating and opening datasets.

Figure 11 demonstrates the performance when the test data is a tensor, with the same data size as Figure 9 and 10. The results are similar as well: HDF5 outperforms others in writing and reading datasets, while netCDF4 is ideal for creating and opening datasets.

From Figure 12 to 14, the number of datasets is doubled, with the number of elements remaining the same. Specifically, what the Figure 12 illustrates matches results from small-scale testing, as well as results when the number of datasets is only 2048: HDF5 is the best for writing and reading data, whereas netCDF4 is fastest in terms of creating and opening datasets.

Figure 13 shows similar results to Figure 12, except writing and reading data in the matrix form. Consistent pattern is found: HDF5 outperforms others in writing and reading datasets, while netCDF4 is ideal for creating and opening datasets. In a similar fashion, Figure 14 reveals identical conclusions in tensor form.

Through Figure 15 to Figure 20, the number of elements becomes even much larger, whereas the number of datasets decrease to 64 and 128. respectively. The result in Figure 15 shows that HDF5 is still the fastest in writing and reading, while netCDF4 remains the slowest.

Figure 16 shows similar results when the random data is transformed from a vector to a matrix, with the same data size in total. HDF5 is ideal for both reading and writing operation, followed by Zarr, and netCDF4 remains the slowest as always.

Figure 17 shows similar results when the random data is transformed from a matrix to a tensor, with the same data size in total. HDF5 is still ideal for both reading and writing operation. However, netCDF4 outperforms Zarr in writing datasets by an average of 100 ms, which is different than pervious results in Figure 3, when the number of elements is relatively small.

Figure 18 shows similar results to Figure 15. HDF5 is still the fastest in writing and reading. However, netCDF4 actually outperforms Zarr in reading datasets this time by an average of 3000ms.

Figure 19 shows similar results to Figure 16. HDF5 is ideal for both reading and writing operation, followed by Zarr, and netCDF4 remains the slowest in both reading and writing performance.

Figure 20 shows similar results to Figure 17. HDF5 is still ideal for both reading and writing operation. However, netCDF4 still outperforms Zarr in writing by 200 ms this time, which is consistent with findings in Figure 17.

Figure 21 shows how each file format performs in writing datasets and reading datasets as the number of elements grows from 1M (1024*1024) to 1G (1024*1024*1024) with 64 datasets. It turns out that HDF5 remains ideal for both reading and writing operation no matter what data size is tested, and netCDF4 in general is not good at writing and reading datasets.

Throughout Figure 22 to Figure 27, the results of compression comparison are demonstrated, where both uncompressed and compressed versions of HDF5, netCDF4, Zarr are tested and compared. Figure 22 shows that netCDF4 is the fastest in creating and opening datasets, regardless of whether it is uncompressed or compressed. Compressed version of HDF5 greatly undermines the performance in creating and opening datasets, whereas no significant difference in creating and opening datasets is found between uncompressed Zarr and compressed Zarr. When it comes to reading datasets,

compressed netCDF4 is the slowest. Both compressed HDF5 and compressed Zarr are taking a huge amount of time in writing datasets. One interesting observation is that blosc-zstd compression does not really affect writing performance for netCDF4 under this specific test case.

Figure 23 shows similar results to Figure 22. Basically, both uncompressed and compressed netCDF4 are the fastest in creating and opening datasets, the slowest in reading datasets. On the other hand, compressed HDF5 and compressed Zarr are much slower in writing datasets.

Figure 24 shows consistent results with Figure 22. Both uncompressed and compressed netCDF4 are the fastest in creating and opening datasets, the slowest in reading datasets. Compressed HDF5 and compressed Zarr are much slower in writing datasets in comparison with uncompressed HDF5 and Zarr.

Figure 25 shows consistent results with Figure 23. Both uncompressed and compressed netCDF4 are the fastest in creating and opening datasets, the slowest in reading datasets. Compressed HDF5 and compressed Zarr are much slower in writing datasets in comparison with uncompressed HDF5 and Zarr.

Figure 26 and Figure 27 demonstrate compression comparison with large-scale data testing. Similar results are found: both uncompressed and compressed netCDF4 are the fastest in creating and opening datasets, the slowest in reading datasets. Both compressed HDF5 and compressed Zarr are taking a huge amount of time in writing datasets. Nevertheless, in comparison with Figure 22, compressed netCDF4 is shown to undermine the writing performance this time, when the data size becomes larger.

Figure 27 shows similar results. Both uncompressed and compressed netCDF4 are the fastest in creating and opening datasets, the slowest in reading datasets. Compressed HDF5, compressed Zarr, and compressed netCDF4 are much slower in writing datasets in comparison with uncompressed counterparts.

Figure 28 to 30 represent how each file format and its compressed version perform in writing datasets and reading datasets as the number of elements grows from 1M (1024*1024) to 100M (1024*1024*100) with 4 datasets. Figure 28 shows that compression largely undermines HDF5 writing performance, and slightly undermines its reading performance.

Almost the same results are justified when is comes to netCDF4. Figure 29 shows that compression greatly undermines netCDF4 writing performance, and slightly undermines its reading performance as the data size grows from 1M (1024*1024) to 100M (1024*1024*100).

Similar results are found for Zarr. Figure 30 shows that compression greatly undermines netCDF4 writing performance, as the data size grows from 1M (1024*1024) to 100M (1024*1024*100). However, compression has no significant influence on Zarr reading performance, which is also illustrated from Figure 22 to Figure 27.

Figure 31, Figure 32, Figure 33 show how compound datatype is tested. Figure 31 shows that CSV outperforms compound HDF5 in both writing datasets and reading datasets in four different fashions when the number of properties/columns is large and the number of elements/rows is small.

Figure 32 shows that with a small number of elements and even a larger number of properties/columns, CSV still outperforms compound HDF5 in both writing datasets and reading datasets in four different fashions.

However, the opposite is true when the number of properies is small and the number of elements is much larger (1M, 1024*1024). Figure 33 shows HDF5 is much faster than CSV in writing datasets and reading datasets by columns/properties. However, no significant difference in reading datasets by rows is found between HDF5 and CSV.

D. Write Benchmark Discussion

The results of this benchmark show that a general trend is that when creating a dataset, netCDF4 takes the least time and is followed by HDF5, which is followed by Zarr. This trend is jusified by a series of large-scale testing and scale element comparison. When it comes to compresssion comparison, compressed HDF5 takes longer than time than uncompressed HDF5, wheres this difference is not found for netCDF4 or Zarr.

When actually writing data to a file, HDF5 takes the least time to write data to a dataset and is followed by Zarr, which is followed by netCDF4—taking on average more than double the time of HDF5. The trend above is well justified by a series of large-scale testing and scale element comparison. When it comes to compression comparison, compressed HDF5 and compressed Zarr are taking a significant amount of time, compared to uncompressed counterparts. Compressed netCDF4 takes much more time than uncompressed netCDF4 when the data size grows large enough. The findings above is justified by scale element comparison as well. When it comes to compound datatype, CSV is much faster in writing datasets when the number of properties is large, while HDF5 is much faster in writing datasets when the number of element grows larger and larger.

E. Read Benchmark Discussion

The read benchmarks show results similar to those from the write benchmark. netCDF4 takes the least time to open a dataset and is followed by Zarr, which is followed by HDF5. The trend above is well justified by a series of large-scale testing and scale element comparison. When it comes to compression comparison, significant difference in opening datasets between uncompressed files and compressed files is not found for HDF5, netCDF4, and Zarr.

When reading the data by printing the dataset values to the standard output, HDF5 takes the least time to read a dataset and is followed by Zarr, which is followed by netCDF4. The trend above is well justified by a series of large-scale testing and scale element comparison. When it comes to compression comparison, compressed file formats are in general slightly slower in reading datasets than uncompressed ones for HDF5, netCDF4, and Zarr. The findings above

is justified by scale element comparison as well. When it comes to compound datatype, CSV is relatively faster in reading datasets when the number of properties is large, while HDF5 is much faster in reading datasets when the number of element grows larger.

V. Conclusions

In this paper, we demonstrated a method in which the performance of a file format can be compared to that of another file format through the running of a benchmark that tests performance in operations like create, open, read, write, and close. This paper focused specifically on benchmarking three file formats: HDF5, netCDF4, and Zarr, as these three file formats are considered to be general-purpose scientific file formats due to their storing of various types of data in a hierarchical manner, similar to a file system.

The benchmark was conducted in Python due to the language's widespread use in numerous scientific applications, and as such, the Python API for each file format was tested. To determine the performance of a file format, the time taken to create a dataset, write data to the dataset, open the dataset once the file is closed, and read data from the dataset was measured and plotted in a bar graph. The results of the benchmark show that HDF5 is fastest in reading or writing to a dataset, netCDF4 is fastest in creating or opening a dataset, and Zarr generally trails rights behind HDF5 in performance. These results are fully tested with large scale data. The effect of compression is also investigated and explored in a preliminary fashion. In general, block compression tends to undermine the file performance of common I/O file operations. Furthermore, HDF5 compound datatype is studied, tested, and compared with CSV in terms of write time and read time. It turns out that the performance depends on the actual shape of the input random data.

Future work for this benchmark would include: expanding support to other programming languages, as this would reveal any potential bottlenecks within the language-specific API for a file format; testing more file formats in order to better determine which file format is the fastest; and testing more aspects of a file format, which may include testing performance in specific scenarios (i.e., reading a small subset of a dataset, overwriting a dataset). Lastly, the code for the benchmark can be found here: `https://github.com/asriniket/File-Format-Testing`. and `https://github.com/Yusen-Peng/SummerResearch2023`.

VI. Acknowledgments

This effort was supported in part by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR) under contract number DE-AC02-05CH11231 with LBNL.

References

[1] CERN. *CERN - Storage*. URL: https://home.cern/science/computing/storage.

[2] NASA. *NASA's Climate Data Services (CDS)*. URL: https://www.nccs.nasa.gov/services/climate-data-services.

[3] Arie Shoshani and Doron Rotem. "Scientific data management. Challenges, technology, and development". In: *Scientific Data Management: Challenges, Technology, and Deployment* (Dec. 2009). DOI: 10.1201/9781420069815.

[4] HDFGroup. *The HDF5® Library & File Format - The HDF Group*. URL: https://www.hdfgroup.org/solutions/hdf5/.

[5] UCAR/Unidata. *Unidata — NetCDF*. URL: https://www.unidata.ucar.edu/software/netcdf/.

[6] CERN. *ROOT: analyzing petabytes of data, scientifically.* URL: https://root.cern/.

[7] Zarr Developers. *Zarr*. URL: https://zarr.readthedocs.io/en/stable/.

[8] Wikipedia. *List of file formats*. URL: https://en.wikipedia.org/wiki/List_of_file_formats#Scientific_data_(data_exchange).

[9] Barbara Warmbein. *Big data takes ROOT*. URL: https://home.cern/news/news/computing/big-data-takes-root.

[10] Cyrille Rossant. *Moving away from HDF5*. URL: https://cyrille.rossant.net/moving-away-hdf5/.

[11] Choonghwan Lee, MuQun Yang, and Ruth Aydt. *NetCDF-4 Performance Report*. URL: https://support.hdfgroup.org/pubs/papers/2008-06_netcdf4_perf_report.pdf.

[12] Leah A. Wasser. *Hierarchical Data Formats - What is HDF5?* URL: https://www.neonscience.org/resources/learning-hub/tutorials/about-hdf5.

[13] HDFGroup. *Introduction to HDF5*. URL: https://portal.hdfgroup.org/display/HDF5/Introduction+to+HDF5.

[14] HDFGroup. *Chapter 3: The HDF5 File*. URL: https://support.hdfgroup.org/HDF5/doc/UG/FmSource/08_TheFile_favicon_test.html.

[15] UCAR/Unidata. *NetCDF4 API Documentation*. URL: https://unidata.github.io/netcdf4-python/.

[16] Zarr Developers. *Zarr*. URL: https://zarr.readthedocs.io/en/stable/tutorial.html#groups/.