Benchmarking for HDF5, What to Expect?

Seminar Thesis
Dlyaver Djebarov

Chair for High Performance Computing, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Supervisor: Radita Liem

With the increasing computational potential of high-performance systems, the amount of data that needs to be analyzed has also increased. For the efficient management of this data, various technologies are used, including the HDF5 technology suite, often used in scientific and development applications. In order to get the best performance out of HDF5, benchmarks are needed, and some of them (IOR, File Format Comparison Benchmark, benchio) are analyzed in this research paper. Special attention is given to h5bench benchmarking and its capabilities, which are tested on the CLAIX-2018 cluster to obtain data on the correlation between write performance, the number of compute nodes and the number of processes on each node, the impact of data dimensionality and I/O patterns, and the read performance of partial and full HDF database.
Keywords: HPC, HDF5 benchmarks, h5bench

1 Introduction

The use of high-performance computing in science and manufacturing inevitably involves the need for efficient I/O that accelerates computation through better scalability and avoids bottlenecks associated with slow memory access. [1] I/O libraries such as HDF5 [2], PnetCDF [3], MPI-IO [4], etc. have been created for this purpose, but not all of them handle real-world tasks equally well and require benchmarking. In this paper, I will focus on looking at HDF5 - the high-performance I/O library for storing and managing heterogeneous data - and its benchmarks. HDF5 has a large community and is often used to develop scientific applications that operate on large amounts of data. Famous examples of HDF5 use in the scientific environment include BioHDF [5], which is focused on storing next-generation sequencing (NGS) data, and HDF-EOS [6], which is used by NASA to archive and distribute remote sensing data from the EOS constellation of satellites. Due to the widespread use of the framework, there is also a need to measure its performance in different configurations and for different supercomputing systems, which is satisfied by numerous benchmarks, ranging from IOR, used to compile the IO500 [7] ranking of high-performance storage systems, to less popular variants like the File Format Comparison Benchmark written in Python.

The beginning of this paper (Section 2) takes a closer look at the principles of HDF5: The way data structures are stored, methods of working with the HDF5 library and their peculiarities, and methods of working with files in parallel. In Section 3, a literature analysis is carried out in which various benchmarks for HDF5 are discussed and compared in detail. The benchmarks reviewed include IOR [8], File Format Comparison Benchmark [9], benchio [10] and h5bench [11]. From the benchmarks discussed in Section 3, I will conduct an in-depth study using h5bench to get a direct insight into the performance of HDF5 in CLAIX-2018 [12]. Section 4 will describe the experimental setup and Section 5 will describe the results of my experiment.

2 HDF5

The HDF5 technology suite is a comprehensive framework comprising a data model, library, and file format designed to store and administer data in high-performance systems. [2] Its architecture allows it to handle many different data types, taking advantage of adaptable and efficient I/O operations to handle complex and large-scale datasets. HDF5 provides portability and scalability, allowing applications to grow with its use. In addition, the suite includes many tools and applications designed to control, modify, visualise and explore data.

2.1 Data structure

Within an HDF5 file, several components align with the HDF5 data model terminology: array variables, groups, and types. These correspond, respectively, to HDF5 datasets, HDF5 groups, and HDF5 datatype objects. HDF5 datasets represent array variables structured as multidimensional arrays, organizing their data elements logically. Each HDF5 dataset is characterized by its dataspace, encapsulating attributes like its rank (dimension count) and the present as well as the maximum size in each dimension. Notably, the HDF5 dataset has the ability to expand and shrink within the limits defined by its maximum size, which can even be unbounded, depending on the storage layout approach.

HDF5 groups function similarly to directories in file systems, where each group explicitly embodies the relationship between different HDF5 information entities, including HDF5 datasets, other HDF5 groups, and HDF5 data type objects. It is important to know that there is a single group in each HDF5 file, denoted as the root HDF5 group, which serves as the underlying entity.

The HDF5 array variable type is based on two main components: the HDF5 data space, which defines its structure, and the HDF5 data type, which defines the nature of its data elements. At the time of writing the paper, the HDF5 array variable type is supported by eleven different data type classes: Integer, Float, String, Bitfield, Time, Opaque, Compound, Reference, Enumeration, Variable Length sequence, and Array. [13] These data types are significant in defining HDF5 datasets and attributes, and may also be associated with HDF5 groups, which in such cases are called HDF5 data objects or HDF5 named data types.

2.2 HDF5 workflow

This section details the operation of the HDF5 library and the basic operations of creating, reading, updating and deleting data. Once an dataset is created, it must be integrated into the HDF5 array database by linking it to at least one HDF5 group. During operations to read or write data to an HDF5 file, the HDF5 library guides the data through a series of processing stages that make up the HDF5 data pipeline (see Figure 1).

This pipeline orchestrates multiple tasks on the data held in memory, including byte swapping, alignment, scatter-gather, and hyperslab selections. Noteworthily, the HDF5 library autonomously determines the necessary operations and controls the execution of memory-related tasks, like extracting specific elements from a data block. Each module within the data pipeline operates on data buffers, sequentially processing and transferring transformed buffers to subsequent stages. The processing stages in the data pipeline are categorised as follows:

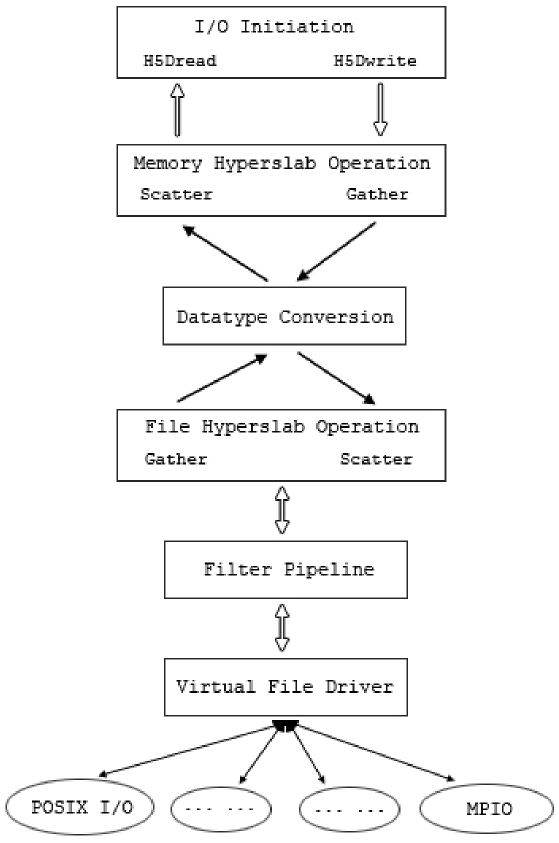

Figure 1: The processing order in the HDF5 data pipeline [14]

- I/O initiation - initiating H5Dwrite and H5Dread in the user programme
- Memory hyperslab operation - data is placed (when reading) or read (when writing) from the memory buffer
- Datatype conversion - data type conversion
- File hyperslab operation - data is placed (when reading) or read (when writing) from file space
- Filter pipeline - if filters are present, data are processed accordingly
- Virtual File Layer - a file driver (e.g. MPIO or POSIX I/O) is used
- Actual I/O - the file driver used by the library (e.g. MPIO or STDIO) is activated

To delete an HDF5 array variable, all connections or links to HDF5 groups must be severed. The released space from this deletion can be recovered provided the HDF5 file remains open in its underlying state.

2.3 Parallel I/O mechanisms

The HDF5 library uses a single global semaphore to protect internal data structures from modification by multiple concurrently executing threads. While this provides thread-safety, it limits simultaneous access to HDF5 files. In MPI-based parallel applications, the HDF5 library extends support for concurrent access to HDF5 files. The main goal is to provide fast parallel I/O operations for extensive HDF5 datasets by leveraging the standardised MPI-IO parallel I/O interface and exploiting the capabilities of parallel file systems such as GPFS [15] and Lustre [16].

Parallel access can be implemented in two modes: independent and collective. In independent mode, individual MPI processes perform I/O operations autonomously, independent of and unaffected by other MPI processes in the same application accessing an identical HDF5 file. In contrast, in collective mode, parallel access implies highly coordinated and cooperative operation of all MPI processes in the application. These two modes differ in programming complexity and have different performance impacts.

Another important feature is that in the context of HDF5, reading and writing are performed in three different modes: synchronous, implicit asynchronous, and explicit asynchronous. [17] In synchronous mode, the I/O phase starts when the computation phase is completed, and the subsequent computation phase starts only after the I/O phase is completed. In contrast, both asynchronous I/O modes, implicit and explicit, allow the subsequent computation phase to begin immediately after I/O operations are passed to background threads. Importantly, in these asynchronous modes, there is no need to wait for data to be written to or read from a file on a storage device; the computation phase begins on its own as soon as I/O operations are initiated in the background.

3 Benchmarking methodology

There are various benchmarking systems that are used to evaluate the performance of HDF5 on modern high-performance systems. Each of these benchmarking systems has its own distinctive features and unique characteristics, which will be discussed in detail in the following subsections.

3.1 IOR

The IOR [8], designed specifically to establish performance benchmarks for the ASCI Purple system procurement at LLNL, focuses on evaluating sequential read and write performance under various file sizes, I/O transaction sizes, and parallelism scenarios. It also provides the ability to use a shared file versus using a dedicated file for each processor. An important feature is compatibility with both conventional POSIX and advanced parallel I/O interfaces, including MPI-IO, HDF5, and parallelNetCDF.

To better understand how benchmarking works, let's take a closer look at the principle of writing to a shared file, illustrated in Figure 2.

In high-level file formats such as HDF5 and NetCDF, each segment corresponds directly to a "dataset" object in the respective file formats. [18] These segments are evenly distributed among the processors sharing the data file, and are organised into "blocks" - contiguous chunks of data accessed by a single client. The blocks are distributed in sequential order: the process with rank 0 receives the first block, the process with rank 1 receives the second block, and so on.

The physical structure of a file reflects application data scattered across distributed memory. Each block is further subdivided into multiple transfer units, referred to as TransferSize. This subdivision is integral for emulating strided access patterns, crucial for reversible decomposition of multidimensional domains.

The TransferSize chunks correspond directly to the I/O transaction size, meaning the amount of data transferred from processor memory to the file each time an I/O function is called, such as the buffer size in a POSIX I/O operation. In a single-file-per-processor scenario, the file structure is very similar to the diagram in Figure 2. However, each process independently writes or reads data to or from its individual file, with the result that each "block" is compactly stored in separate files.

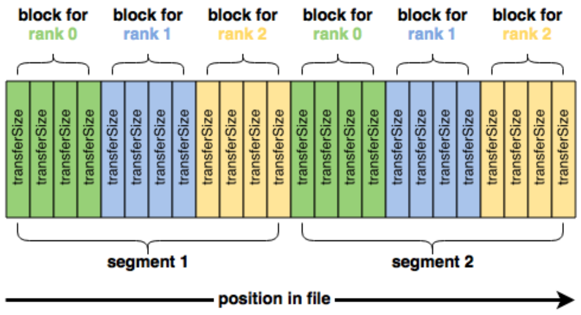

Figure 2: The design of the IOR write benchmark for shared files [19]

When performing a read-after-write benchmark, it is fundamentally important to consider the possible caching of data in DRAM (dynamic random access memory). [19] During the write phase of the IOR, data can be accidentally cached because Linux uses a write-back cache to buffer I/O. Instead of the IOR interacting directly with the file system when writing and reading data, it primarily interacts with memory on each compute node. In particular, although each IOR process thinks of itself as writing to a file on the file system and then reading its contents, it actually does:

- Writing data to a cached copy of a file in memory. If no cached copy exists before writing, the changed parts are loaded into memory.
- Marking modified parts of a file in memory (called "pages") as "dirty".
- Completing the write() call in IOR, even if the written data is not yet committed to the file system.
- Independently, the OS kernel periodically scans the file cache for files that have been updated in memory but are not in the file system (i.e., dirty pages). It then commits the modifications to the file system cache, declaring dirty pages to be non-dirty because they are now synchronized to disk, even though they remain in memory.

Subsequently, during the read phase of the IOR, the application can retrieve the contents of the file directly from memory, avoiding the need to communicate with the file system over the network.

To evaluate the read performance of the underlying file system, one approach is to intentionally exceed the total page cache capacity during the write phase. This ensures that by the time the write phase is complete, the initial sections of the file have already been fetched from the cache, thus providing a more realistic estimate of read performance.

3.2 File Format Comparison Benchmark

The Python-based File Format Comparison Benchmark, detailed in a research paper by its creators [9], evaluates the time it takes to perform three key operations: creating a data set, writing data, and then reading data from the created data set at a later time. These operations fall into the fundamental categories of writing and reading, which are critical to evaluating the efficiency of a file format, especially when storing data for long-term processing.

Write and read speeds are a good indicator of the characteristics of a high-performance system and have a significant impact on the end-user workflow by minimizing time spent on operations unrelated to the main task.

This benchmark uses a configuration-based system, allowing users to set test parameters, such as the number of datasets to create in a file and the array size for each dataset, by editing the YAML configuration file. The main stated feature is the benchmark's methodology after writing data to a file: it renames and moves the file to a different directory to reduce the effect of caching, which can affect subsequent read times.

To test read operations, the benchmark opens a file, reads all the data sets in it, and outputs them to the standard output. This process of executing the write operations benchmark and the subsequent read operations benchmark is repeated several times to ensure consistency and reliability of the collected data.

3.3 benchio

Another way to perform parallel I/O performance evaluation in HDF5 is to use the benchio application developed at EPCC, which has several advantages over such tools. [10] First, benchio allows for a flexible parallel I/O decomposition that better matches real user application models. In contrast, IOR uses a rudimentary one-dimensional data decomposition that is not sufficiently representative of user codes and does not allow performance evaluation of the crucial MPI-IO collective operations that significantly affect real-world performance evaluation.

Furthermore, benchio is characterized by its code transparency, which allows for a better understanding of performance variations compared to the relatively opaque nature of IOR. Another notable advantage is benchio's ability to evaluate the performance of both HDF5 and NetCDF, which broadens its scope of application.

In particular, benchio estimates the write throughput to a single shared file based on a given task size per processor (so-called weak scaling), with the output file size increasing in proportion to the number of processors. Test data is structured as a series of double-precision floating-point numbers organized into a 3D array distributed across processes using 3D block decomposition. For more accurate modeling of real scientific applications, halos are introduced by default in all dimensions of the 128^3 local array.

Halos act as boundary layers surrounding the main computational domain in arrays or grids. [20] These additional layers contain duplicate or mirrored data from neighboring areas, forming a bridge for information exchange between neighboring cells or elements. Halos serve as critical interfaces to ensure seamless communication and computation between interconnected regions. Their importance is to improve modeling accuracy by ensuring the smooth flow of data and computation across these boundaries.

By preserving relevant information from neighboring regions, these boundary layers simplify computations by ensuring that interactions and dependencies with surrounding elements are accurately accounted for. This functionality plays a key role in a variety of scientific applications, as it allows for more efficient and accurate simulations by properly accounting for the effects of neighboring elements on the main computational domain.

3.4 h5bench

The following benchmark is discussed more thoroughly and extensively as it is used to dive deeper into the topic and it is the one that will be used later on for experiments to get the performance results of HDF5 for the CLAIX-2018 computational system.

3.4.1 h5bench I/O kernel suite

The development of h5bench [11] was undertaken with the goal of creating a complete set of HDF5 I/O kernels that adequately represent applications using the HDF5 API, while optimizing their I/O performance by leveraging the new features introduced in HDF5.

Also used in h5bench are the write (VPIC-IO) and read (BD-CATS-IO) cores. VPIC [21] is a carefully optimized and scalable particle physics simulator developed at Los Alamos National Laboratory. It was originally derived from a plasma physics simulator designed to study magnetic reconnection phenomena that are often found in space weather events such as solar flares interacting with Earth's magnetosphere. VPIC-IO [22] uses the H5Part API to create a file, write eight variables, and then close the file. The H5Part API serves as a simplified interface for executing HDF5 calls corresponding to the time-varying multivariate particle data model. The VPIC-IO core encapsulates all H5Part function calls extracted from the VPIC code by targeting an array of one-dimensional particles, each with eight variables. The particle data generated in the kernel consists of random float point data.

BD-CATS-IO [23], on the other hand, originates from the parallel DBSCAN (Density-based spatial clustering of applications with noise) algorithm, specifically adapted to read particle data like those generated in VPIC simulations.

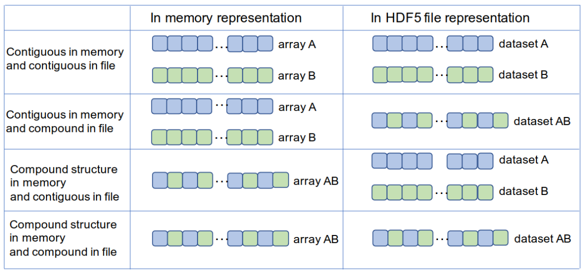
In memory representation
In HDF5 file representation
Contiguous in memory and contiguous in file
array A
array B
dataset A
dataset B
Contiguous in memory and compound in file
array A
array B
dataset AB
Compound structure in memory and contiguous in file
array AB
dataset A
dataset B
Compound structure in memory and compound in file
array AB
dataset AB

Figure 3: In-memory data structure and in HDF5 file data layout mappings

The design of the h5bench benchmark suite assumes that the simulation or analysis is performed over many time steps, including several subsequent computation and I/O phases. This design follows typical patterns observed in physics simulations, which involve extensive computation of numbers over many time steps to emulate the physical phenomenon under investigation. The state of the simulation is often recorded in memory, either to examine the progress of the simulation or to provide a checkpoint to eliminate possible failures.

No real computation is performed in the benchmark suite, instead using sleep() to emulate computation time. The data generated in the write benchmarks is randomized and the current time is used as a seed. The read benchmarks then use the data generated in the write benchmarks and emulate the data analysis time using sleep(). The duration of the emulated computation time can be specified in the configuration file, which provides flexibility for users in the write and read benchmarks.

3.4.2 Locality

With the expansion of the write and read benchmarks, h5bench is not limited to memory and file locality. The benchmark suite has been augmented with a variety of I/O modes and patterns, including asynchronous I/O, multidimensional arrays, file system specific configurations, and MPI-IO specific configurations. Users are provided with customizable options in h5bench, allowing them to use different I/O patterns according to their requirements.

To better understand I/O patterns, let's look at the ways of storing data in memory and in an HDF5 file. A contiguous in-memory pattern involves multiple arrays, all of which contain elements of the same basic data type, such as integer, float, double, etc. In contrast, the non-contiguous in-memory pattern, also known as "array of structures" or "derived data type", involves data types derived from the underlying data types. HDF5 supports storing both arrays of base data types and derived data types. When storing arrays of base data types, each array is stored as a separate data set. On the other hand, derived datatypes are stored in HDF5 as a "compound datatype" dataset. This is illustrated more clearly in Figure 3.

3.4.3 I/O modes

H5bench supports write and read operations with different templates. The benchmark is designed to define HDF5 datasets and read or write metadata associated with those datasets. Metadata includes important information such as dataset sizes and names.

The fundamental HDF5 write and read APIs used in h5bench are H5Dwrite and H5Dread, responsible for writing to and reading from HDF5 datasets, respectively. Both write and read benchmarks adhere, as in benchio, to weak scaling, a configuration in which the number of particles per rank is set by the user via a configuration file. As the number of MPI ranks increases, the data size increases proportionally, creating a scalable representation of the computational load.

In h5bench, the MPI ranks adopt non-overlapping partitions of HDF5 datasets through hyperslab selections in HDF5. The benchmark provides support for three distinct types of hyperslab selections: contiguous, strided, and partial.

In the contiguous access pattern, each MPI rank accesses one continuous partition of the data. In contrast, the strided pattern involves hyperslab selections with interspersed gaps, and the stride is specified in the configuration file. The stride can be defined across multiple dimensions, allowing for a varied representation of data access patterns.

The partial pattern is exclusive to read operations, where analysis applications read only a subset of the data. This corresponds to scenarios where analysis applications typically focus on specific subsets of the data rather than the entire dataset.

Another feature of h5bench is the support of three modes for both read and write: synchronous, implicit asynchronous and explicit asynchronous. These modes were described in detail in Section 2.3.

In synchronous mode, the I/O phase starts when the calculation phase is completed. Subsequently, the next computation phase starts only after the completion of the I/O phase. In contrast, the two asynchronous I/O modes, explicit and implicit, differ from this synchronous model. In these modes, the subsequent computation phase begins after I/O operations in background threads are started, without waiting for data to be fully written to a file or read from a file on a storage device. This asynchronous approach improves efficiency by executing the computation and I/O phases simultaneously, potentially reducing overall execution time.

Arrays with multiple dimensions are the predominant data structures in scientific datasets. h5bench, recognizing this multidimensional nature, makes it also easy to read and write datasets with dimensions from 1D to 3D. Users can specify these and the parameters described above in a configuration file to customise the benchmarking process to meet their specific requirements. A typical configuration file looks like the one shown in Figure 4.

3.4.4 Running h5bench

The JSON configuration file for h5bench includes five main sections: mpi, vol, file-system, directory, and benchmarks. Each property must be defined, even if its body is empty. Configuration allows users to specify the MPI startup programme (e.g. mpirun, mpiexec, srun) and the desired number of processes to execute. For more complex configurations or precise control over job parameters, users can define a "configuration" property that h5bench will use when running experiments using the provided "command" property. If a configuration parameter is specified, h5bench will ignore the "ranks" property.

```
{
  "mpi": {
      "command": "mpiexec",
    "ranks": "384"
  },
  "vol": {
    "library": "/vol-async/src",
    "path": "/vol-async/src",
    "connector": "async under_vol=0;under_info={}"
  },
  "file-system": {
    "lustre": {
      "stripe-size": "1M",
      "stripe-count": "4"
    }
  },
  "directory": "storage",
  "benchmarks": [
    {
      "benchmark": "write",
      "file": "test.h5",
      "configuration": {
        "MEM_PATTERN": "CONTIG",
        "FILE_PATTERN": "CONTIG",
        "TIMESTEPS": "5",
        "DELAYED_CLOSE_TIMESTEPS": "2",
        "COLLECTIVE_DATA": "YES",
        "COLLECTIVE_METADATA": "YES",
        "EMULATED_COMPUTE_TIME_PER_TIMESTEP": "1 s",
        "NUM_DIMS": "1",
        "DIM_1": "4194304",
        "DIM_2": "1",
        "DIM_3": "1",
        "CSV_FILE": "output.csv",
        "MODE": "SYNC"
      }
    }
  ]
}
```

Figure 4: Example of a h5bench configuration file

H5bench supports the use of HDF5 VOL connectors [24], such as async and cache, for h5bench_write and h5bench_read. However, since some benchmarks in h5bench may not support VOL connectors, users need to include the relevant information in the configuration file to control the VOL setup at runtime. To do this, specify absolute paths to the required libraries using the "library" property, specify the path to the VOL connector and configure it in the "connector" property. The above example shows the configuration of the HDF5 asynchronous VOL connector.

H5bench creates a special catalogue for each workflow, where all generated files and logs are stored. Additional options, such as data striping for Lustre, can be applied to this directory if configured. Users can use the "file-system" property to configure specific file system options. Currently, this property can be used for Lustre to define the number and size of striping for a catalogue storing data generated by h5bench.

The "benchmarks" property allows users to specify which benchmarks h5bench should run, their order and configuration options. The following benchmarks are available to choose from: write, write unrestricted, overwrite, add, read, metadata, exerciser, openpmd, amrex, e3sm. For each h5bench template, the user needs to specify the appropriate file name and configuration. If the same filename is specified as the one used for the previous write, h5bench will read from that file, allowing workflows to be configured with multiple interleaved files (e.g. write file-01, write file-02, read file-02, read file-01).

To run the benchmark, the following command must be executed:

```
h5bench configuration.json
```

After the benchmark finishes its work after some time, a directory with the files stdout, stderr and output.csv will appear in the directory specified in the configuration file. The first two contain standard output and standard error output respectively. The third file has the following structure, which may vary slightly depending on the benchmark parameters:

```
metric, value, unit
operation, write,
ranks, 64,
collective data, YES,
collective meta, YES,
subfiling, NO,
total compute time, 4.000, seconds
total size, 40.000, GB
raw time, 2152.177, seconds
raw rate, 19.040, MB/s
metadata time, 0.001, seconds
observed rate, 18.613, MB/s
observed time, 2204.507, seconds
```

From this file you can get all the results of the test, including the run time and performance of the tested operation.

4 Experimental setup

I evaluated h5bench on the CLAIX-2018 [12] cluster at RWTH Aachen University in order to get more data on HDF5 performance, namely, I looked at write performance for different numbers of nodes and cores on them in NFS and BeeOND file systems, read performance for part of a dataset and the whole dataset, the impact of data locality on read performance, and the impact of the dimensionality of the stored data on write performance.

The h5bench (version v.1.4) benchmark of cluster performance was run via Slurm batch system with the following parameters:

- –**nodes** - the number of nodes used. Varies in each test from 1 to 16
- –**ntasks-per-node** - number of tasks per node. In the tests on writing perfomance from 6 to 48 processes were used, while in the tests on the read perfomance and the impact of locality and shape 16 processes per node were used.
- –**mem-per-cpu** - the amount of memory per CPU. In all tests the maximum memory of 3900 Mb was requested
- –**beeond** - flag for using the BeeOND file system. It was used in tests on write perfomance.

The number of iterations used in each test is 5, sleep after each iteration to emulate computation lasts for 2 timesteps. The write and read data, except for the dimensionality impact test, are one-dimensional and contain 4,194,304 values. Used HDF5 version HDF5/1.14.0 is loaded in sbatch scripts with the command **module load HDF5/1.14.0**. To run MPI in h5bench, the Intel MPI environment and the **mpiexec** command were utilized.

4.1 CLAIX-2018

Benchmarking was conducted on the CLAIX-2018 cluster located in Aachen, Germany at RWTH Aachen University and established in 2018. The cluster consists of 1297 nodes, 1243 of which contain 2 Intel Xeon Platinum 8160 Processors "SkyLake" with a clock speed of 2.1 GHz and 24 cores each.

Thus there are 48 cores per node and 192 GB of main memory (4 GB of memory per core). The total theoretical peak performance is 4009 TFLops. All computing nodes in CLAIX-2018 are connected to an Intel Omni-Path 100G network, yielding a latency between nodes of 0.93 μsec and a bandwidth of 93 GBit/s.

4.2 NFS

Network File System (NFS) [25] is a distributed file system protocol that enables remote file access and sharing on a network. Created by Sun Microsystems in the 1980s, NFS has become the standard for file sharing in Unix and Linux environments. NFS works on a client-server model, with a central server hosting shared directories, allowing multiple client machines to interact with files as if they were local. This system simplifies the management of files and resources on a network. The simplicity and efficiency of NFS make it a popular choice for transparent access to files and directories on interconnected machines. Although the NFS protocol has been enhanced in newer versions to address security and performance issues, its underlying principles of simplicity and network file sharing still underlie its widespread use in a variety of computing environments.

4.3 BeeOND

One of the available file systems on the cluster is BeeGFS [26]. It has the advantage that it is a parallel file system and is designed to work efficiently with storage nodes. The high speed of data access is mainly due to the file storage approach, where files are segmented into chunks of a certain size, which are then distributed to different storage servers. Metadata is also distributed at the directory level to different metadata servers, often integrated with storage servers.

BeeOND is designed to more flexibly exploit parallel file system capabilities during a single computational task by creating a temporary file system shared by multiple nodes. Another advantage of using BeeOND is that applications running on it do not affect other global file system processes in any way, which also works the other way round: the entire capacity of BeeOND disks can be used only for the task at hand, without interference from others. In the tested CLAIX-2018 cluster, BeeOND is already integrated with the Slurm [27] workload manager.

5 Perfomance evaluation

When performing the tests, all measurements were performed 3 times and the maximum was always chosen as the result, similar to the methodology described in the original paper [11]. In the name of avoiding caching effects the tests were not performed consecutively. In most cases the observed I/O rate - the amount of data written/read divided by the time spent on the corresponding operations, measured in MB/s, was chosen as the result, but the tests for full and partial file reading also use elapsed time in seconds.

5.1 Write perfomance

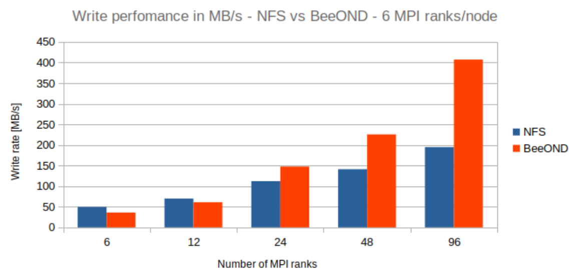

Figure 5: Comparison of synchronous write performance with different number of computational nodes for 6 MPI ranks per each for NFS and BeeOND.

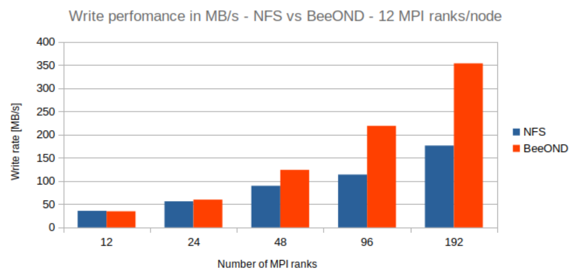

Figure 6: Comparison of synchronous write performance with different number of computational nodes for 12 MPI ranks per each for NFS and BeeOND.

In Figures 5, 6, 7, 8, 9, 10 I compared the write performance when using NFS and BeeOND as file systems for 1 to 16 compute nodes and 6, 12, 16, 24, 32, and 48 processes on each. The careful reader will notice that the graphs for 32 and 48 prouesses per node are missing two columns for NFS. This is because when using h5bench to test write performance, an HDF5 file is created to which data

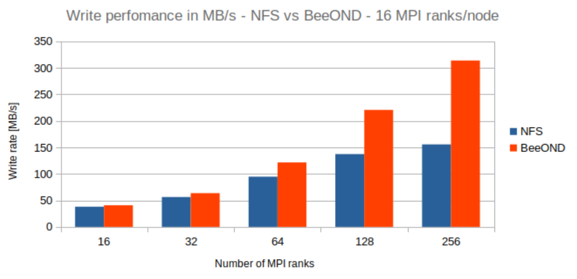

Figure 7: Comparison of synchronous write performance with different number of computational nodes for 16 MPI ranks per each for NFS and BeeOND.

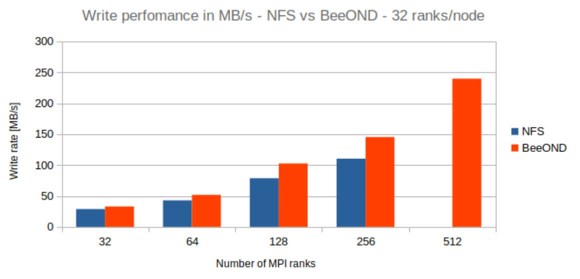

Figure 9: Comparison of synchronous write performance with different number of computational nodes for 32 MPI ranks per each for NFS and BeeOND.

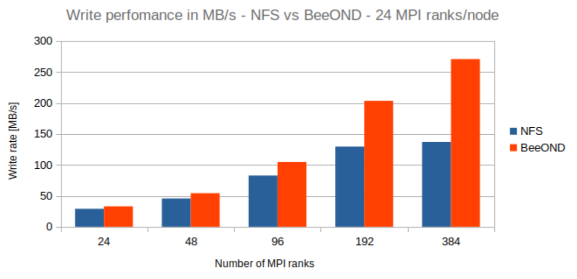

Figure 8: Comparison of synchronous write performance with different number of computational nodes for 24 MPI ranks per each for NFS and BeeOND.

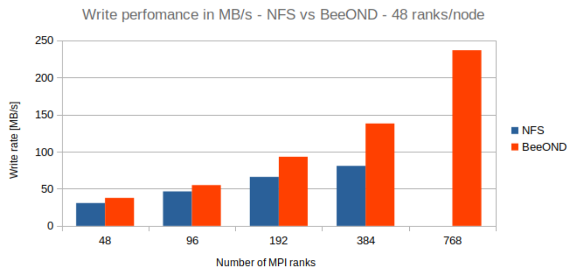

Figure 10: Comparison of synchronous write performance with different number of computational nodes for 48 MPI ranks per each for NFS and BeeOND.

is written throughout the test. Since the tests are weak scaling, the file size grows in proportion to the number of processes, which for 384 and 768 processes is over 250 Gb, which is the quota limit per user in the cluster. For this reason I was not able to get complete data for 32 and 48 processes per node. In spite of this, we can see an obvious trend of increasing write bandwidth with increasing number of nodes in use. Also the write performance in BeeOND is on average 1.3 times higher. For the experiment with 6 processes per node for 1 and 2 nodes it can be seen that BeeOND performance is even worse than NFS. This is due to the fact that parallel file systems have overhead when dealing with small file sizes. [28]

It is also noteworthy that when using more MPI ranks per node, the overall performance of the system degrades. As it turned out, the problem was in choosing Intel MPI as the environment, because Intel-mpiexec uses only one core instead of the one specified in the batch script, thus competing for 1 core resource, due to an identified problem on the cluster that I was unaware of when doing the measurements. By choosing OpenMPI environment and testing with srun, I got better performance and the anomaly disappeared. So the performance degradation when the number of processes increases is due to the fact that due to weak scale testing the amount of data to write increases but due to a bug on the cluster, more processes are not allocated and thus more work is done on the single process.

5.2 Impact of locality

As already described in section 3.4, the benchmark has the ability to test write performance for different combinations of I/O patterns and the following were chosen for this test:

- contiguous in memory and contiguous in file
- contiguous in memory and compound in file
- compound in memory and contiguous in file
- compound in memory and compound in file

Each is encoded in the graph legend as CC, CI, IC and II respectively. For the tests, 1 to 16 nodes with 16 processes on each node were used. The results are shown in Figure 3 and it can be seen that, as in the original paper, maximum performance is achieved when using matching layouts in memory and in the file (CC and II). In the other two cases, due to the conversion of individual arrays to HDF compound datatype (CI) and arrays of structures in memory to individual HDF5 datasets (IC), there is a dramatic drop in performance. I was also able to confirm the observation that when writing individual arrays in memory to the HDF5 compound datatypes file (CI), the worst performance is achieved. Another interesting observation is that writing a view of individual arrays in memory to individual HDF5 datatypes (CC) loses to writing structures from memory to HDF5 compound datatypes (II) by almost a factor of two. This may be due to the fact that configuration files from the original paper, made specifically for CORI (a Cray XC40 system at The National Energy Research Scientific Computing Center (NERSC)) and which may not be suitable for testing the memory mapping inside CLAIX, were used for the test. Another possible cause is described in section 5.1 and may be due to incorrect operation of mpiexec.

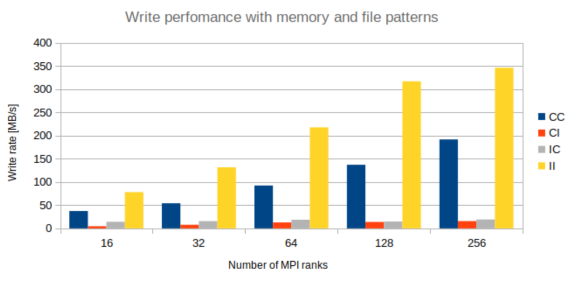

Figure 11: Observed write rate for different I/O patterns and different number of computational nodes with 16 MPI ranks per each.

5.3 Impact of the shape of arrays

In Figure 12, I have presented the write performance for arrays of different dimensions using 1 to 16 nodes with 16 MPI ranks per node. In all cases the arrays consist of 262144 elements. Thus the two-dimensional array is represented as 512 x 512 elements and the three-dimensional array is 64 x 64 x 64. Arrays of structures or compound data types are not used in the test. Also, as in the original paper, I have shown that the write performance for data of different dimensionality is the same. At first glance, it may seem that greater performance is achieved by using 256 MPI ranks for 1D arrays (18.132 MB/s) relative to 2D arrays (17.263 MB/s) and 3D arrays (17.317 MB/s), but this difference of 0.869 MB/s and 0.815 MB/s is an error inevitable under the condition of cluster workload.

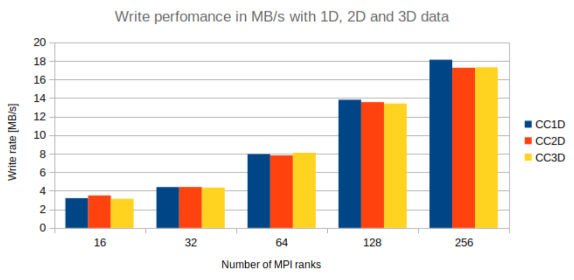

Figure 12: Observed write rate of different array sizes (1D, 2D and 3D) written with different number of computational nodes for 16 MPI ranks per each.

5.4 Full and partial reading performance

The result of the synchronous read performance test is shown in Figure 13 and Figure 14. In this test, the HDF5 file was first created and then either a full read or a read of the first 10% of the contents was performed. As in the previous case, measurements were performed on 1 to 16 nodes with 16 processes on each node. It is easy to notice that the time of reading the whole dataset is about 100 seconds and does not change when changing the number of MPI ranks involved. As in the original paper, this is due to the fact that the test is weak scaling. In other words, when the number of involved processes increases, the amount of data to read also increases. However, it turned out that the same conclusion is not true for the time of reading a part of the file. The graph shows that the time to read a part of the file increases with the number of processes involved. Also on the second chart it can be seen how much the performance of reading a part of a file slows down compared to full reading. This could be caused by the mpiexec problem described in Section 5.1.

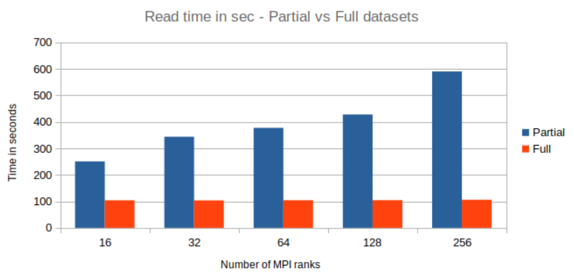

Figure 13: Comparison of the observed time to read the entire dataset and reading its 10% at different number of computational nodes for 16 MPI ranks on each.

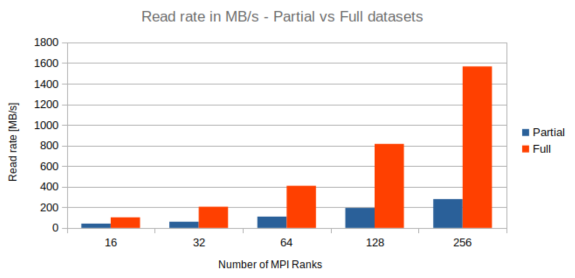

Figure 14: Comparison of the observed read rate by reading the entire dataset and its 10% at different number of computational nodes for 16 MPI ranks on each.

6 Conclusion and future work

In this paper, I presented an overview of the actual benchmarks created to determine the performance of HDF5 in the context of high performance computing. After describing the basic principles of HDF5 and getting an idea of data storage, HDF5 library methods and parallel file handling, I reviewed and analyzed benchmarks IOR, File Format Comparison Benchmark, benchio and h5bench, which I used to take a deep dive on the actual use case and measure the performance of the CLAIX-2018 cluster in terms of the correlation between write performance, number of compute nodes, number of processes on each node, file system selection, data dimensionality, I/O patterns, and read performance of partial and full HDF5 databases. Because of the described problem with the Intel MPI environment, the data I have presented cannot be considered reliable, but even so, some trends are still true, for example, the conclusion that the dimensionality of the data written to the HDF5 file does not affect the write performance or the observation that converting arrays of structures in memory into individual HDF5 datasets and individual arrays in memory into HDF5 compound datatype leads to a significant performance decrease. In the future, I plan to conduct tests using OpenMPI environment and srun utility, which will give more accurate results.

References

[1] Adrian Jackson et al. “High performance i/o”. In: *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*. IEEE. 2011, pp. 349–356.

[2] Mike Folk et al. “An overview of the HDF5 technology suite and its applications”. In: *Proceedings of the EDBT/ICDT 2011 workshop on array databases*. 2011, pp. 36–47.

[3] Rob Latham et al. “The parallel-netCDF I”. In: *O Library. Argonne National Laboratory* (2010).

[4] Rajeev Thakur, William Gropp, and Ewing Lusk. “On implementing MPI-IO portably and with high performance”. In: *Proceedings of the sixth workshop on I/O in parallel and distributed systems*. 1999, pp. 23–32.

[5] Christopher E Mason et al. “Standardizing the next generation of bioinformatics software development with BioHDF (HDF5)”. In: *Advances in Computational Biology*. Springer. 2010, pp. 693–700.

[6] Ilg Doug et al. *HDF-EOS Library User’s Guide for the ECS Project, Volume 1: Overview and Examples*. Tech. rep. Raytheon Systems Company (Upper Marlboro, Maryland), 1999.

[7] *IO500*. 2023. URL: `https://io500.org/about` (visited on 01/16/2024).

[8] *IOR Benchmark Repository*. 2024. URL: `https://github.com/hpc/ior` (visited on 01/16/2024).

[9] Sriniket Ambatipudi and Suren Byna. “A comparison of hdf5, zarr, and netcdf4 in performing common i/o operations”. In: *arXiv preprint arXiv:2207.09503* (2022).

[10] B Lawrence et al. *Parallel I/O performance benchmarking and investigation on multiple HPC architectures*. 2017.

[11] Tonglin Li et al. “h5bench: HDF5 I/O kernel suite for exercising HPC I/O patterns”. In: *Proceedings of Cray User Group Meeting, CUG*. Vol. 2021. 2021.

[12] RWTH Aachen IT Center. *RWTH High Perfomance Computing: Hardware overview*. 2023. URL: `https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/fbd107191cf14c4b8307f44f545cf68a/` (visited on 07/12/2023).

[13] The HDF Group. *HDF5: API Reference v1.14.3*. 2023. URL: `https://docs.hdfgroup.org/hdf5/v1_14` (visited on 01/08/2024).

[14] The HDF Group. *HDF5 User’s Guide*. 2011. URL: `https://davis.lbl.gov/Manuals/HDF5-1.8.7/UG/UG_frame10Datasets.html` (visited on 01/08/2024).

[15] Jason Barkes et al. “GPFS: a parallel file system”. In: *IBM International Technical Support Organization* (1998).

[16] *Official website of the Lustre® filesystem.* 2024. URL: `https://www.lustre.org/` (visited on 01/16/2024).

[17] Houjun Tang et al. "Transparent asynchronous parallel I/O using background threads". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2021), pp. 891–902.

[18] Hongzhang Shan and John Shalf. *Using IOR to analyze the I/O performance for HPC platforms.* Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2007.

[19] *IOR Documentation.* 2023. URL: `https://ior.readthedocs.io/en/latest/userDoc/tutorial.html` (visited on 01/16/2024).

[20] Siegfried Benkner. "Optimizing irregular HPF applications using halos". In: *International Parallel Processing Symposium.* Springer. 1999, pp. 1015–1024.

[21] Surendra Byna et al. "Parallel I/O, analysis, and visualization of a trillion particle simulation". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE. 2012, pp. 1–12.

[22] Kesheng Wu, Surendra Byna, and Bin Dong. *Vpic io utilities.* Tech. rep. Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2018.

[23] Md Mostofa Ali Patwary et al. "BD-CATS: big data clustering at trillion particle scale". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 2015, pp. 1–12.

[24] The HDF Group. *The HDF5 Virtual Object Layer (VOL).* 2023. URL: `https://docs.hdfgroup.org/hdf5/develop/_h5_v_l__u_g.html` (visited on 01/18/2024).

[25] Brian Pawlowski et al. "The NFS version 4 protocol". In: *In Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000.* 2000.

[26] Frank Herold, Sven Breuner, and Jan Heichler. "An introduction to BeeGFS". In: *ThinkParQ, Kaiserslautern, Germany, Tech. Rep* (2014).

[27] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing.* Springer. 2003, pp. 44–60.

[28] Marcin Krotkiewski. "Dealing with small files in HPC environments: automatic loop-back mounting of disk images". In: *Partnership for Advanced Computing in Europe Report.* 2017.