Diese Arbeit wurde vorgelegt am
Lehrstuhl für Hochleistungsrechnen (Informatik 12), IT Center.

Erweiterung von h5bench um I/O-Zugriffsmuster in gängigen KI-Anwendungen

Extending h5bench with I/O access patterns in common AI applications

Bachelorarbeit

Dlyaver Djebarov
Matrikelnummer: 423901

Aachen, den 14. Oktober 2024

Erstgutachter:	Prof. Dr. rer. nat. Matthias S. Müller (')
Zweitgutachter:	Prof. Dr. Sarah Neuwirth (*)
Betreuer:	Radita Liem, M.Sc (')

(') Lehrstuhl für Hochleistungsrechnen, RWTH Aachen University
IT Center, RWTH Aachen University
(*) Johannes Gutenberg University of Mainz

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen sind, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form noch nicht als Prüfungsarbeit eingereicht worden.

Aachen, den 14. Oktober 2024

Kurzfassung

Die rasche Einführung von Künstlicher Intelligenz (KI) in der wissenschaftlichen Datenverarbeitung erfordert neue Werkzeuge zur effektiven Bewertung der I/O-Leistung. HDF5 ist eines der Datenformate, die nicht nur in HPC, sondern auch in modernen KI-Anwendungen häufig verwendet werden. Die bestehenden Benchmarks sind jedoch unzureichend, um die aktuellen Herausforderungen von KI-Workloads zu bewältigen. In diesem Beitrag wird eine Erweiterung des bestehenden HDF5-Benchmarks - h5bench - vorgestellt, indem die gleiche Arbeitslast aus dem MLPerf Storage - DLIO Benchmark integriert wird. Diese Erweiterung ermöglicht es Nutzern, KI-Workloads zu testen, ohne dass sie Bibliotheken für maschinelles Lernen installieren müssen, was die Komplexität reduziert und die Nutzbarkeit des Benchmarks verbessert. Die experimentelle Analyse zeigt, dass die Erweiterung in der Lage ist, die bestehenden I/O-Muster mit leicht anpassbaren Konfigurationen zu replizieren, um verschiedene Skalierbarkeitstests durchzuführen.

Stichwörter: HPC, Benchmarking, HDF5, I/O, I/O Benchmark, I/O Kernel, h5bench, AI Workloads, MLPerf

Abstract

Rapid artificial intelligence (AI) adoption in scientific computing requires new tools to evaluate I/O performance effectively. HDF5 is one of the data formats commonly used not only in HPC applications but also in modern AI applications. However, the existing benchmarks are insufficient to address the current challenges posed by AI workloads. This thesis introduces an extension to the existing HDF5 benchmark, called h5bench, by incorporating the same workload from the MLPerf Storage - DLIO Benchmark. This extension allows users to test AI workloads without the need to install machine learning libraries, reducing complexity and enhancing the usability of the benchmark. The experimental analysis demonstrates that the extension managed to replicate the existing I/O patterns with easy-to-adjust configurations to perform various scalability tests.

Keywords: HPC, benchmarking, HDF5, I/O, I/O benchmark, I/O kernel, h5bench, AI workloads, MLPerf

Contents

List of Figures

List of Tables

1. Introduction

Artificial intelligence (AI) has been swiftly integrated into scientific computing, driving significant breakthroughs across various disciplines. [58] [25] [5] As AI usage grows, so does the demand for I/O systems capable of supporting these increasingly complex workloads. Multiple new special systems are currently being developed to accommodate this emerging AI workload [74][24] since the I/O performance bottleneck problem is getting more pronounced. Despite the critical impact of I/O performance in AI applications, there is a limited number of benchmarks specifically targeting this problem, with Deep Learning I/O (DLIO) Benchmark [28] now the leading benchmark for AI workloads, and it is the kernel of the MLPerf Storage [9] benchmark suite.

Besides the lack of options in the current I/O benchmarking landscape for AI workloads, a number of further issues have been identified that present significant challenges to the evaluation of current systems. First, the existing AI benchmarks require the installation of machine learning libraries that are often not directly related to the metrics that will be evaluated, such as bandwidth and throughput scalability. Setting up the correct environment for running AI experiments is one of the productivity challenges faced by developers and performance analysts for evaluating AI workloads [63]. This challenge that looks trivial at first glance can create unnecessary complexity and overhead, making it difficult to isolate and evaluate the I/O characteristics critical for AI workloads. Another challenge arises from the fact that most HPC systems serve both AI workloads and traditional HPC applications simultaneously. Setting up a system that can accommodate both workloads is an essential task for every HPC center's stakeholders. The result from the AI-focused benchmark needs to have the same context as the result from the benchmark from the more traditional workloads to produce the right conclusion for decision-making.

To address these challenges, I propose extending the existing I/O benchmark suite, h5bench [13], to better accommodate AI workloads. Hierarchical Data Format version 5 (HDF5) [80] is one of the data formats that is commonly used not only in HPC applications [15] and has great potential for use in AI applications [27] [12]. h5bench already features a configuration structure suited for testing traditional HPC workloads, and this framework can be extended to handle AI workload scenarios as well. In the initial prototype, an investigation was conducted into DLIO workloads with the objective of incorporating the specific configuration requirements essential for AI applications into the h5bench extension.

This thesis begins by outlining the key challenges in benchmarking I/O components for AI workloads in Chapter 2, followed by a detailed exploration of HDF5, existing I/O benchmarks, AI workloads in general, and profiling tools used to gather I/O patterns in Chapter 3. In the Chapter 4, I describe in detail the methods by which I/O patterns generated by DLIO Benchmark were obtained and discuss how to design a system based on them. This chapter also includes thoughts on how to validate the implementation. Chapter 5 contains an information about the main features and the most important details of the h5bench extension implementation. The experimental setup and the results are presented in detail in Chapter 6. The limitations of the current work and future work are discussed in Chapters 7 and 8.

2. Related work

Among the existing tools for I/O benchmarking, IOR [1] stands out as the leading parallel I/O benchmark, widely used to evaluate the bandwidth performance of parallel file systems. IOR supports a variety of file access patterns, such as read-/write operations, shared files, and file-per-process setups. It is compatible with multiple I/O library interfaces (POSIX, MPI-IO, HDF5) and also integrates with distributed computing systems like S3 and HDFS. The IOR code repository also contains MDTest [57], which is used to assess the metadata performance of POSIX-compliant parallel file systems. MDTest measures the performance of file, directory, and directory hierarchy operations, including creation, stat, and removal. Users may use IOR and MDTest to devise a multitude of I/O patterns, as seen in the IO500 [53] benchmark suite, making this benchmark very powerful but also overwhelming for users.

In addition to h5bench [13] which is employed in this thesis and will be described in detail in Section 3.3, numerous other HDF5 benchmarks exist that are based on the application's patterns. These have been previously listed in the work of Bez et al.[13]. For example, MACSio [61] (Multi-purpose, Application-Centric, Scalable I/O Proxy Application), that has been developed to address a long-standing gap in the co-design of proxy applications that test I/O performance and evaluate trade-offs in data models, I/O library interfaces, and parallel I/O paradigms for multiphysics HPC applications. Or AMRex [93] - a framework for massively parallel block-structured adaptive mesh refinement (AMR) utilised in a range of scientific disciplines, including combustion, accelerator physics, carbon capture and storage, cosmology and astrophysics, whose access scheme can be classified as contiguous in memory and contiguous in file representation. Other prominent members of this list are Parallel I/O Kernel Suite [17], FLASH-IO [47] and ChomboIO [21].

In the context of AI applications and machine learning (ML) applications, with a particular focus on performance in HPC systems, MLPerf [69] is one of the prominent benchmarks in the field. Organized by the MLCommons working group [2], MLPerf offers a variety of applications that employ machine learning and deep learning techniques within the context of HPC environments. MLPerf encompasses a multitude of focal areas, including Inference, Training, and Storage. DLIO benchmark [28] is an important part of the MLPerf for Storage [9] benchmark.

Fathom [4] was developed to study the characteristics of cutting-edge models from the deep learning community. The collection consists of eight archetypal deep learning workloads, including three focused on solving image classification tasks using different models [44] [71] [52]. The remaining workloads represent models for direct language-to-language sentence translation [75], memory-oriented neural

system [86], speech recognition engine [43], autoencoder for dimensionality reduction and image compression [46], and Atari-playing neural network from DeepMind [62].

In the broader context of ML benchmarking, there are several other benchmarks listed by Thiyagalingam et al. [81], such as Deep500 [11], RLBench [48], DAWNBench [22], and AIBench [34].

Considerable work has been done to describe the data-centric nature of ML applications by Noah Lewis et al. [54] This includes detailed descriptions of file formats, common phases of the workflow, and an overview of I/O benchmarks and profilers. This is important because the use of benchmarks plays an important role in finding I/O bottlenecks and subsequently achieving greater efficiency. A section of the paper focuses on I/O optimisation techniques for popular machine learning libraries, highlighting their strengths and weaknesses. The paper concludes with an outlook on future research in this area, making it an excellent overview of current trends in I/O for machine learning on HPC systems.

3. Background

3.1. Brief ML overview

Since the h5bench extension aims to simulate AI workloads to measure I/O performance, it is essential to provide a brief overview of the typical machine learning (ML) workflow and the unique aspects of working with data and data formats. This information is outlined in the following sections.

3.1.1. Workflow

A typical machine learning workflow involves four key phases: data generation, dataset preparation, model training, and inference [65]. Figure 3.1, the result of the work of Noah Lewis et al. [54], provides a complete taxonomy of the operations and data management characteristics of these stages. The data generation phase involves acquiring data from sources such as simulations, web scraping, or publicly available datasets. Critical decisions are made at this stage regarding the dataset's modality, file format, and storage location.

Next is the dataset preparation phase, which involves steps such as transforming, augmenting, and splitting the data. In the training phase, the taxonomy outlines the input-output operations linked to commonly used learning algorithms, along with checkpointing tasks that preserve model states.

The output phase mirrors the training phase in terms of data operations, involving similar data management processes. Overall, the taxonomy emphasizes the central role of data throughout the ML lifecycle, underscoring its importance at every stage of model development and deployment. This close relationship between data and machine learning highlights the importance of efficient I/O management, which is discussed in the following section.

3.1.2. I/O in ML

In machine learning, I/O operations are crucial because they dictate how efficiently data flows between storage and the computing resources that process it. As machine learning models grow in complexity and datasets expand, managing I/O performance becomes increasingly critical. This is especially true during training phases, where large volumes of data must be read continuously, placing a heavy load on storage systems. One algorithm that exemplifies the I/O challenges in machine learning is gradient descent.

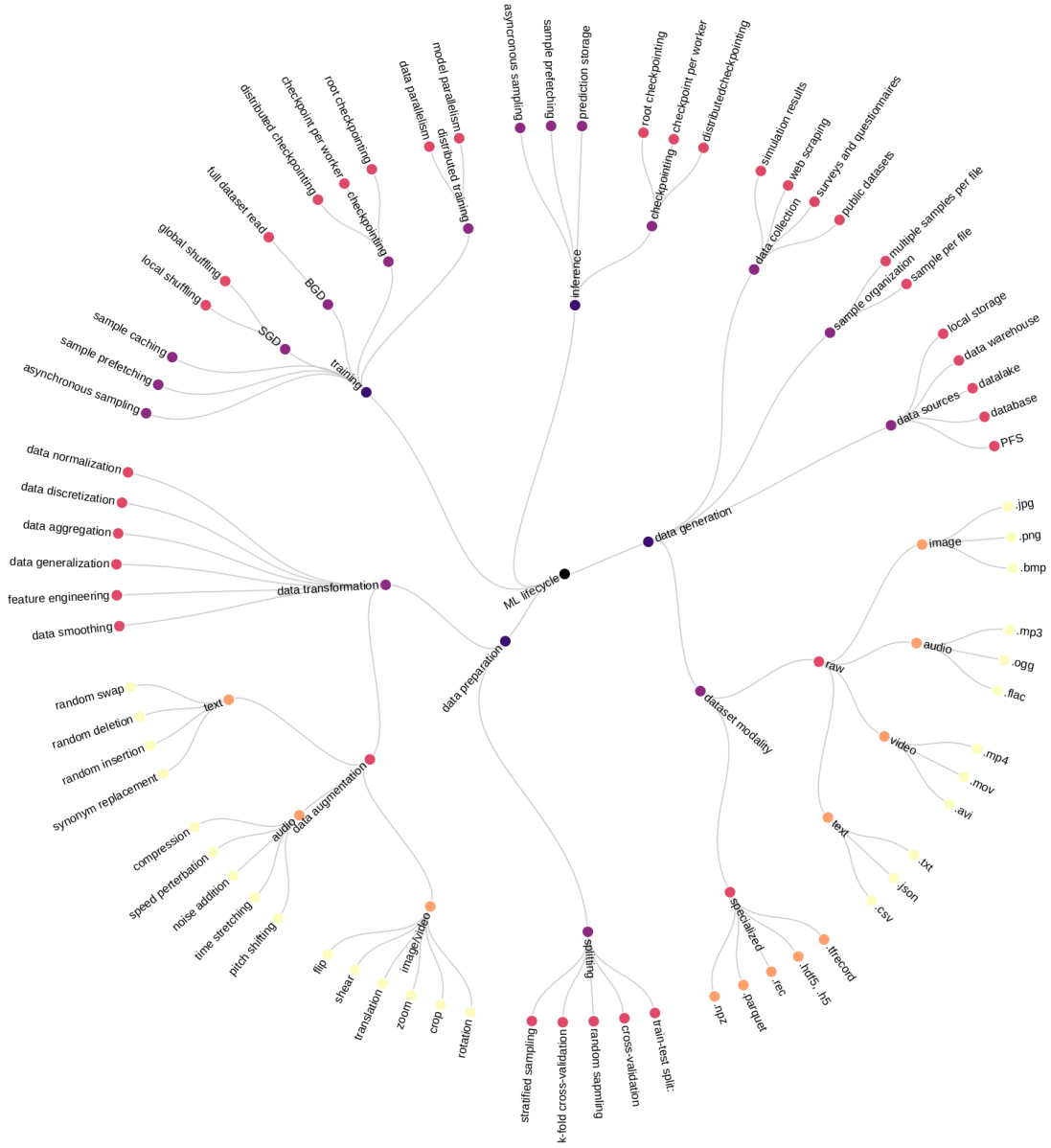

Figure 3.1.: A taxonomy of data management during the ML lifecycle. [54]

Gradient descent, particularly mini-batch gradient descent - commonly known as stochastic gradient descent (SGD) - is one of the most widely used algorithms for optimizing deep neural network (DNN) parameters. Its high learning speed and low memory consumption make it a preferred choice. Nevertheless, despite these advantages, there are several challenges associated with the efficient utilisation of storage resources.

Increasing mini-batch size

An increase in the mini-batch size may be beneficial. The success of SGD has prompted the development of numerous optimisations designed to capitalise on the increasing computational power. One key strategy is to increase the mini-batch size during each training iteration, allowing for the simultaneous processing of more input samples. It has been demonstrated in numerous studies [90] [60] [91] that significant improvements can be made in terms of learning speed. In these studies, the mini-batch size has increased from 256 [44] to 80.000 [90] over the past few years. However, increasing the mini-batch size requires a corresponding boost in throughput, as more samples must be processed with each iteration. Furthermore, the utilisation of substantial batch sizes has the potential to impair the quality of the model. This is due to the fact that methods employing large batch sizes tend to converge to sharp minimizers of the training objective function, which can have a detrimental impact on the model's generalisation capabilities. [50]

Small random samples

A further challenge in the training of deep learning models is the necessity to handle a large number of training samples, which often arrive in a random order. [67] The randomness inherent in such data helps to speed up convergence and reduce errors caused by fixed input sequences. However, many datasets consist of many small samples. [20] This poses storage problems, as small random samples often result in random read operations, which do not work well in traditional storage systems optimised for large sequential I/O operations.

Pre-processing smaller samples into larger ones can alleviate issues with small random I/O operations, but this approach comes with its own limitations. For instance, extant sample shuffling techniques are incapable of supporting global shuffling of data, and the size of the shuffling buffer further constrains the efficiency of shuffling. [89] To illustrate, when utilising TFRecord files, each file is read sequentially into a fixed-size shuffle buffer. If the buffer size is insufficient, only a proportion of the samples will be shuffled, which may have a detrimental impact on the accuracy of the training process. Therefore, the size of the shuffle buffer represents a crucial bottleneck in guaranteeing the efficient shuffling of samples and the maintenance of optimal model performance.

3.1.3. Data formats

As data represents the most crucial element of any machine learning workload, the choice of file formats for the storage of training and inference data is of equal importance. In the context of HPC, it is crucial that the file format employed is capable of supporting efficient distributed data storage. Notable examples of such formats include HDF5 [80], RecordIO [7] and Apache Parquet [8]. HDF5 and RecordIO, in particular, offer unique features such as chunking and partial compression

mechanisms. These will be discussed in greater detail in section 3. Both features permit data loaders to write and read only subsets of a file, negating the necessity to fully load them. Given that ML applications frequently read numerous small data fragments, this represents a significant advantage for the aforementioned formats. Furthermore, the capacity to compress subsets of data enables more efficient data storage, which may result in faster I/O by reducing the overhead associated with transferring data while it is being read by the ML program. In the next sections, the HDF5 file format will be discussed in detail.

3.2. HDF5

HDF5 [80] is a comprehensive framework that includes a data model, library, and file format. It is designed for storing and managing data in high-performance systems. The HDF5 architecture efficiently handles various data types using adaptable I/O operations, making it ideal for managing large and complex datasets. HDF5 also provides portability and scalability, allowing applications to grow with user needs. [32] Additionally, the suite includes a range of tools and applications for controlling, modifying, visualizing, and exploring data.

Section 3.2.1 provides an overview of HDF5's hierarchical structure, which is its defining characteristic (as reflected in its name: Hierarchical Data Format v5). Section 3.2.2 discusses the parallel version of HDF5, widely used in high-performance systems and integral to my h5bench extension. The following Section 3.2.3 will describe the HDF5 workflow, which is the key to understanding concepts such as Virtual File Driver (VFD) and Virtual Object Layer (VOL), described in more detail in Section 3.2.4 and Section 3.2.5, respectively. Section 3.2.6 explains the key features of HDF5, such as chunking and compression, which can greatly impact the performance of ML applications. While these features are included in my extension, they are not available in the original DLIO Benchmark.

3.2.1. Data structure

HDF5 files contain components aligned with the HDF5 data model terminology, including array variables, groups, and types, which correspond to HDF5 datasets, groups, and data type objects, respectively. Datasets represent multidimensional arrays with logically organized data elements. Each HDF5 dataset is defined by its data space, which encapsulates attributes such as its rank (dimension count) and the present as well as the maximum size in each dimension. It is noteworthy that the HDF5 dataset is capable of expansion and contraction within the limits defined by its maximum size, which may even be unbounded, depending on the storage layout approach.

HDF5 groups function like directories in a file system, defining relationships between HDF5 datasets, other groups, and data type objects. Every HDF5 file contains a root group, which serves as the top-level entity.

The HDF5 array variable type is based on two principal 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 HDF5 array variable type is supported by eleven different data type classes, as follows: Integer, Float, String, Bitfield, Time, Opaque, Compound, Reference, Enumeration, Variable Length sequence, and Array. [42] 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.

3.2.2. Parallel HDF5

To fully exploit the potential of parallelism in high-performance computing, HDF5 offers a specialized configuration known as Parallel HDF5. This configuration uses the Message Passing Interface (MPI) standard for interprocess communication and incorporates the standard parallel I/O interface (MPI-IO)[79]. One of the key benefits of this approach is that HDF5 files created in parallel mode remain compatible with serial HDF5, as they conform to the established HDF5 file format specification [41]. The Parallel HDF5 API comprises two major components: the standard HDF5 API and a parallel "wrapper". This wrapper opens files via the MPI Communicator, allowing different files to be opened with different communicators. Notably, parallel access requires synchronization across all MPI processes, with operations coordinated in accordance with the collective Parallel HDF5 API. This includes essential tasks such as modifying namespace and structural metadata. Collective I/O consolidates multiple small I/O operations into fewer, larger ones. In independent mode, individual MPI processes perform I/O operations autonomously, without interference from other MPI processes in the same application accessing an identical HDF5 file.

To enable Parallel HDF5 during the compilation process, it is merely necessary to use the optional `--enable-parallel` flag. When creating an HDF5 file in collective mode, users must configure a file access property to regulate how each MPI process accesses the file. This is achieved through the function call `H5Pset_fapl_mpio()`, which takes MPI Communicator and MPI Info objects as parameters. Importantly, no further configuration is required for basic operations such as creating, opening, or closing a dataset, as these tasks are performed collectively by all processes within the MPI Communicator by default.

Regarding dataset access, users are granted considerable flexibility in determining how write and read operations are conducted. The necessary configuration can be achieved through the dataset transfer property, which can be accessed via the function `H5Pset_dxpl_mpio()`. In this way, Parallel HDF5 seamlessly integrates with MPI to optimize data access in parallel environments. Additionally, it retains backward compatibility with traditional serial HDF5, ensuring that files created in parallel can be used in serial workflows without modification.

3.2.3. HDF5 workflow

When reading or writing data to an HDF5 file, the HDF5 library manages the data's flow through a series of stages called the HDF5 data pipeline (see Figure 3.2). This pipeline handles various operations on data in memory, including byte swapping, alignment, scatter-gather, and hyperslab selections. It is noteworthy that the HDF5 library autonomously determines the necessary operations and controls the execution of memory-related tasks, such as the extraction of specific elements from a data block. Each module in the data pipeline processes data buffers sequentially, transferring them to the next stage. The process begins with the initiation of I/O via the H5Dwrite and H5Dread functions, responsible for writing and reading data in the user program. The processing stages in the data pipeline are categorised as follows:

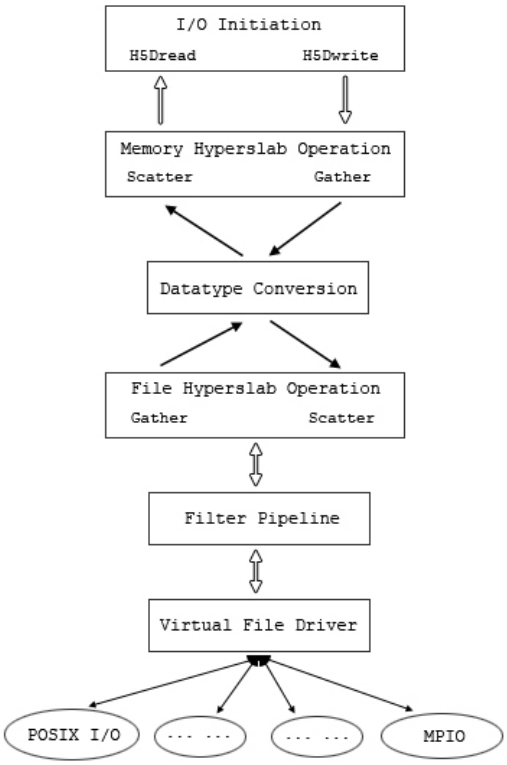

Figure 3.2.: The processing order in the HDF5 data pipeline. [37]

- Operation of memory hyperslab — A hyperslab is a portion of a dataset, which can be a contiguous block of points or a regular pattern in the data space. At this stage, data is placed (for reading) or retrieved (for writing) from the memory buffer.
- Datatype conversion — In certain instances, it may be necessary to convert datatypes in accordance with their storage in the file and memory. One example of such a conversion is the conversion from big-endian to little-endian and vice versa.
- File hyperslab operation: This stage is analogous to the memory hyperslab operation stage, with the sole distinction being that data is now placed (during reading) or read (during writing) from the file buffer.
- Filter pipeline — HDF5 permits the transmission of data in the form of chunks through filters that have been defined by the user and arranged in a pipeline, either to or from the disk. One example of such filters is zlib [33], a generalisation of the DEFLATE data compression algorithm used in the gzip data compressor.

- Virtual File Layer — Virtual file drivers manage the transfer of data between memory and disk. More details on these drivers can be found in Section 3.2.4.
- Actual I/O — At this stage, the file driver employed by the library (e.g. MPIO or STDIO) is initiated.

In order to delete an HDF5 array variable, it is necessary to sever all connections or links to HDF5 groups. The space released by this deletion can be recovered, provided that the HDF5 file remains open in its underlying state.

3.2.4. Virtual File Drivers

HDF5 maps internal data structures and source data to a linear address space via the HDF5 library's API. However, the HDF5 specification does not dictate how this address space is translated into physical storage. In earlier HDF5 versions, the entire address space was mapped to a single file.

As user needs evolved, the HDF5 address space had to accommodate various storage options, such as single or multiple files, local or distributed storage, and network protocols. This flexibility addresses multiple use cases. To illustrate, some users are required to work with exceptionally large data sets within operating systems that impose limitations on file size. In such cases, partitioning the address space into equal-sized segments, with each stored in a separate file, is an effective solution. Others seek to optimise I/O performance by dividing the address space by data type (e.g. source data, object headers, global heap) and storing each type in a separate file.

As the specific storage configurations that may be required are inherently unpredictable due to the vast array of potential scenarios, a virtual file-level API has been developed. The aforementioned API permits HDF5 users to design and implement custom mappings between the HDF5 address space and underlying storage. Such mappings assume the form of individual file drivers, which may be constructed upon existing drivers, thus affording enhanced flexibility. Figure 3.3 illustrates the position of VFDs within the HDF5 hierarchy.

Subfiling VFD

One illustrative example of such a driver is the HDF5 Subfiling Virtual File Driver (Subfiling VFD) introduced in version 1.14.0. [45]. This MPI-based driver allows HDF5 files to be distributed into smaller subfiles. Each subfile is constituted of data segments of equal size, designated as "stripes". The subfiling VFD works with a system of "I/O Concentrators" to direct I/O operations to the appropriate subfile in a configurable pattern.

The subfiling VFD permits the user to define specific parameters, including the dimensions of the data segment and the number of subfiles. In this manner, the user is able to achieve an equilibrium between the two well-established parallel I/O methodologies: the utilisation of a single shared file and the implementation of a

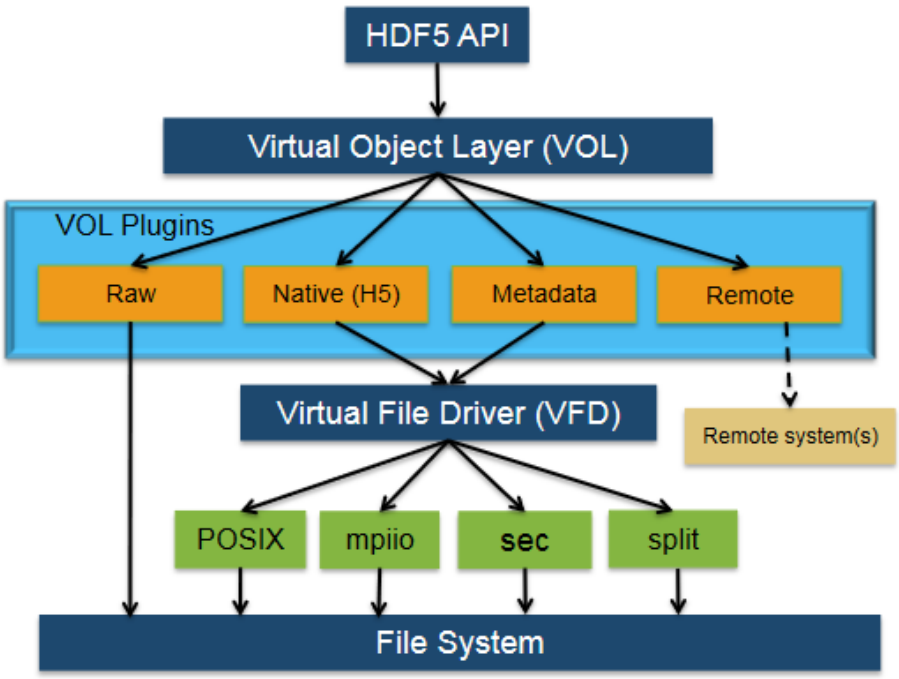

Figure 3.3.: HDF5 architecture and the position of VFD and VOL within it. [14]

file-to-process approach, with the objective of optimising the performance of the specific machine on which the HDF5 application is running.

The subfiling functionality is based on two principal components: the Subfiling VFD itself and the underlying I/O Concentrator VFD. The Subfiling VFD is responsible for the management of configuration data and metadata that are specific to the subfiling process. Furthermore, it divides I/O requests into suitable offset-length pairs in accordance with the established subfiling parameters. Subsequently, these requests are conveyed to the I/O Concentrator VFD, which is situated beneath the Subfiling VFD within the virtual file driver stack, as illustrated in Figure 3.4. The I/O Concentrator VFD is responsible for receiving offset-length pairs from the Subfiling VFD and directing each I/O request to the appropriate I/O concentrator (typically an MPI rank, dedicated thread, etc.) that manages the appropriate section of the HDF5 logical file.

When an HDF5 file is created with Subfiling VFD, multiple files are generated. The first is a stub file, resembling a standard HDF5 file but containing only a small portion of the metadata. This file is used by HDF5 to ascertain when the file was created with the Subfiling VFD. The second file is a configuration file, which defines the Subfiling settings. Finally, the individual subfiles are created.

Log VFD

Log VFD [38] (distinct from Log VOL) is another VFD designed to ensure that file access is occurring correctly and to log HDF5 file access patterns, detailed infor-

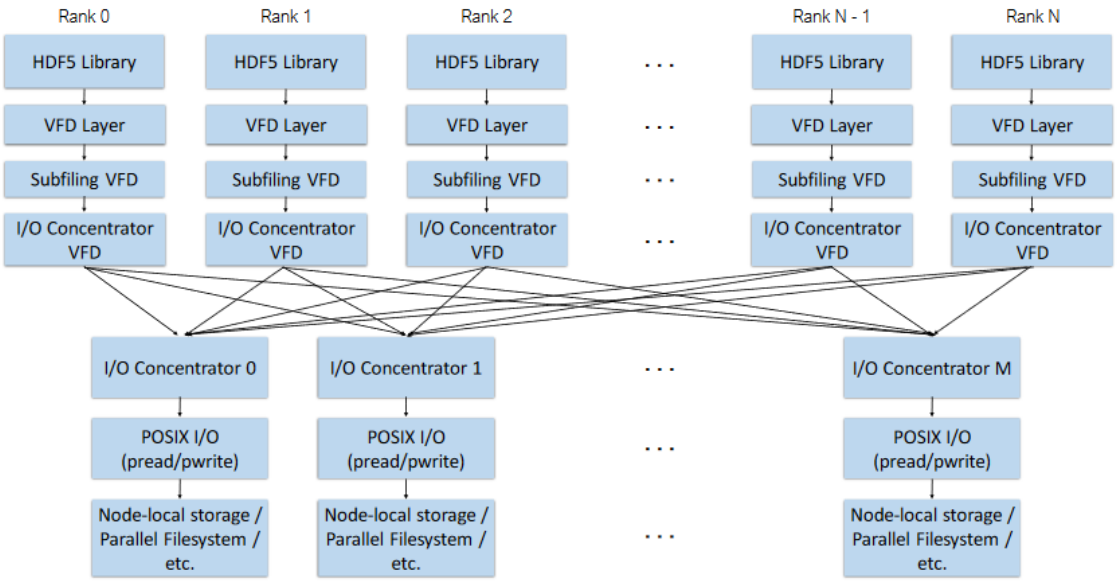

Figure 3.4.: Subfiling VFD architecture. [45]

mation about file operations, memory allocations, dataset reads and writes. This information will be used in Section 4.1.2 to analyse I/O patterns in ML. The Log VFD is capable of functioning in real time, obviating the necessity for recompilation of the application or recreation of the HDF5 file with which it is to be employed. It is sufficient to set the environment variable `HDF5_DRIVER` to the value log in order to enable the necessary functionality. The default output is directed to the standard error stream, where it appears as a list of records in the format:

```
16-         95 (         80 bytes) (H5FD_MEM_SUPER) Read
```

IThe first two digits represent the range of bytes being accessed or allocated in the HDF5 file, followed by the byte count in parentheses. Next, the type of memory accessed is shown, as in the above example, where it's indicated as superblock memory (`H5FD_MEM_SUPER`). In addition, the type of operation performed on the file fragment is indicated, including whether it was read, written to, or allocated. Furthermore, it is possible to modify the file access property within the source code by utilising the `H5Pset_fapl_log()` function.

3.2.5. Virtual Object Layer

The Virtual Object Layer (VOL) in the HDF5 library acts as an abstraction layer, intercepting HDF5 API calls that access objects in the HDF5 container and performing operations like caching, mirroring, and more. These calls are then forwarded to the VOL connector, which performs the actual storage operations through the aforementioned VFD. This enables users and applications to employ the familiar HDF5 data model and APIs, and to translate the physical storage of HDF5 files

and objects into storage systems that are better aligned with the data requirements of the application.

Located directly beneath the public API (see Figure 3.3 for details) the VOL initiates a series of fundamental input verification processes when a storage-associated public API call is initiated. The library then calls upon a VOL callback that corresponds with the selected VOL connector implementation, which was designated at the time the file was initially opened or created. Subsequently, the VOL connector executes the requisite operations, after which control is returned to the library. At this stage, any concluding operations, such as the assignment of identifiers to newly created or opened datasets, are conducted prior to the function's return. This structure implies that for calls that utilise VOL, the majority of the functionality is managed by the VOL connector, with the HDF5 library performing only a limited set of tasks. It thus follows that the majority of HDF5 caching layers (e.g. metadata and chunk caches, page buffering, etc.) are not accessible, given that they are implemented in the native VOL connector for HDF5 and cannot be readily exploited by external connectors.

It is important to note that not all public HDF5 API calls are routed through VOL - only those calls that are related to storage manipulation. This necessitates the implementation of an appropriate callback by the author of the VOL connector. Calls pertaining to data space, property lists, error stacks, and so forth are not associated with storage manipulation or queries and, as a consequence, do not utilise VOL.

Furthermore, it is important to note that not all VOL connectors will support the entirety of the public HDF5 API. It should be noted that the development of certain features may not yet have been completed, or that there may not be an equivalent in the target storage system. Moreover, a considerable number of the HDF5 public API calls are particular to the native HDF5 file format, and therefore may not be applicable to alternative VOL connectors.

Async VOL Connector

The HDF5 Asynchronous I/O VOL connector [77] [76] was designed with the objective of supporting asynchronous I/O operations. This can result in a notable reduction in I/O time and the overall performance of data-intensive applications, due to the fact that I/O operations can be overlapped with computation and communication processes. In write applications, write operations do not overlap at any stage; in analysis applications, they only overlap at the initial read stage. Therefore, the observed I/O overhead is almost negligible if the overhead of managing asynchronous I/O is minimal.

Figure 3.5 illustrates the operational principle of the framework. UWhen asynchronous I/O is enabled, a background thread is automatically started for each application process. It intercepts all I/O operations and creates corresponding asynchronous tasks. These tasks are stored in a queue for dependency evaluation and subsequent execution. The background thread is continuously monitoring the status

of the main thread of the application. It initiates the execution of accumulated tasks only when it detects that the application is no longer engaged in I/O operations. Upon termination of the application, the asynchronous I/O framework ensures the completion of all outstanding I/O operations, releases the allocated resources, and terminates the background thread.

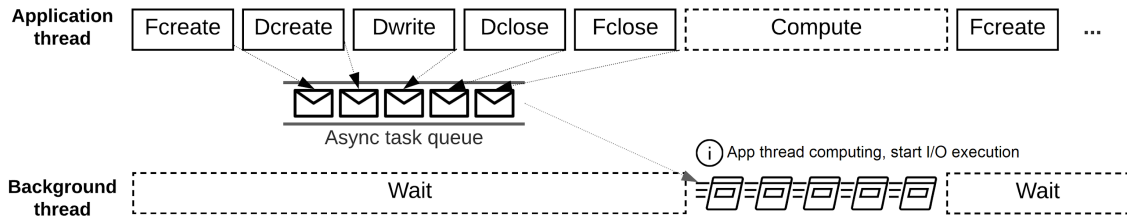

Figure 3.5.: Demonstration of how the Async VOL Connector works. [77]

Two ways of using asynchronous I/O are implemented: implicit and explicit modes. Implicit mode requires minimal code changes, but has performance limitations. To use it, it's only need to set the `HDF5_PLUGIN_PATH` and `HDF5_VOL_CONNECTOR` environment variables. Unlike implicit mode, explicit mode requires some code changes, but allows to take full advantage of asynchronous execution by grouping asynchronous I/O operations into EventSets.

3.2.6. Chunking and compression

In the HDF5 format, datasets may be represented as multidimensional arrays with up to 32 dimensions. [40] Nevertheless, within an HDF5 file, these datasets are required to be stored as part of a one-dimensional data stream, which is referred to as a low-level file structure. The process of mapping a multidimensional dataset into a sequential data stream is referred to as its layout. The most straightforward approach to achieve this mapping is to flatten the dataset in a manner analogous to the storage of arrays in memory. This entails serialising the entire dataset into a continuous block on disk, which can then be directly mapped to a memory buffer corresponding to the dataset's size. This layout method is referred to as the contiguous layout.

Another approach is the chunked layout. Unlike contiguous datasets, which are stored as a single large entity, chunked datasets are divided into smaller units stored in different locations within the HDF5 file. These chunks can be placed arbitrarily without a specific sequence. One of the principal advantages of chunking is the capacity to read or write chunks independently, which can notably enhance performance, particularly when working with subsets of a dataset rather than the entire dataset.

The division of chunks in a partitioned dataset is based on logical boundaries within the dataset's array structure, as opposed to their position in the serialised file. By selecting the appropriate chunk size, it is possible to optimise I/O performance

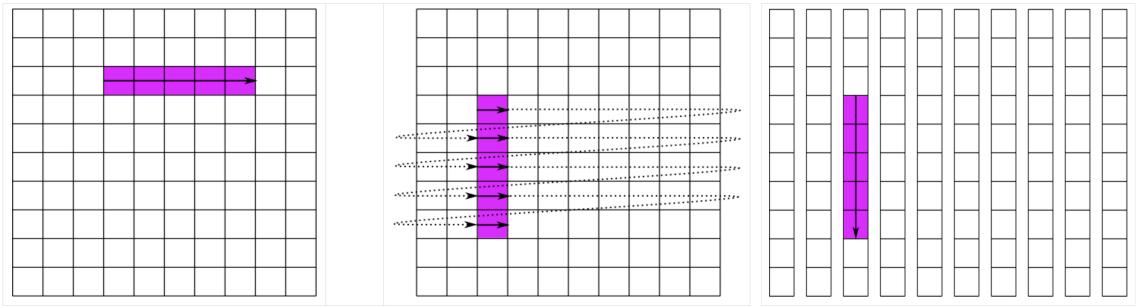

Figure 3.6.: Reading a column of data in both a contiguous and chunked layouts.

for a range of access patterns (see example in Figure 3.6). The capacity to modify the chunk size enables users to adapt the data layout to suit particular applications, thus guaranteeing efficient reading and writing.

A further crucial element of the chunking strategy is the chunk cache, which serves the function of storing chunks of the dataset within memory. The chunk cache has the potential to notably enhance performance when the same chunks are accessed on multiple occasions, as it can reduce the necessity for the HDF5 library to repeatedly read from or write to the disk. However, the current implementation of the chunk cache requires manual adjustment of its parameters to achieve optimal performance, as it lacks the capacity to automatically adapt to usage patterns. The chunk cache is maintained for each open dataset, and thus exists as a discrete entity. Consequently, the caching behaviour is specific to the dataset in question.

Another important element of chunking is the chunk cache, which stores chunks of the dataset in memory. The chunk cache can significantly improve performance when the same chunks are accessed multiple times, reducing the need for the HDF5 library to repeatedly read from or write to disk. It is therefore crucial to select appropriate chunk sizes and cache settings in order to achieve optimal performance, particularly when utilising filtered datasets. The calibration of these parameters in an appropriate manner ensures the efficient processing and storage of data, particularly in scenarios where compression or other I/O filters are employed.

3.3. h5bench

h5bench [55] [13] s an HDF5 benchmark suite developed to provide a comprehensive set of I/O kernels that accurately reflect the functionality of applications using the HDF5 API. It also aims to enhance I/O performance by leveraging the latest HDF5 features, such as VOL connectors and VFDs.

The following sections discuss the baseline I/O modules used in h5bench to assess the performance of read and write operations on the target system. In my implementation, these modules serve as reference points for the results generated, and their outputs and metrics will also be reviewed. I will also describe the architecture and workflow of the h5bench benchmark suite, along with a detailed explanation of the

Benchmark	SYNC	Async VOL	Cache VOL	Log VOL
h5bench write	yes	yes	yes	yes
h5bench read	yes	yes	yes	yes
Metadata stress	yes	no	no	no
AMReX	yes	yes	no	no
Excerciser	yes	no	no	no
OpenMD write	yes	no	no	no
OpenMD read	yes	no	no	no
E3SM-IO	yes	no	no	yes
MACSio	yes	no	no	yes
DLIO	yes	no	no	no

Table 3.1.: Benchmarks available in h5bench

configuration files used. These files follow a standard format across all benchmarks, including the one I developed, with only minor differences in parameter names.

3.3.1. I/O access patterns

The flexible modular system allows for the addition of new modules representing various I/O patterns. Currently, nine modules are available (details in Table 3.1), but this thesis focuses on the first two baseline I/O modules introduced in h5bench.

The primary modules entail read and write benchmarks based on the existing VPIC-IO and BD-CATS-IO kernels. VPIC [16] is a scalable particle physics simulator developed at Los Alamos National Laboratory. It was originally derived from a plasma physics simulator used to study magnetic reconnection phenomena, such as those found in space weather events like solar flares interacting with Earth's magnetosphere. The VPIC-IO [87] module generates a file, writes eight variables, and then terminates the file. The particle data generated in the kernel is composed of random floating-point data.

In contrast, BD-CATS-IO [66] has its origins in the parallel DBSCAN (Density-based spatial clustering of applications with noise) algorithm, which has been specifically adapted to read particle data generated in VPIC simulations.

The h5bench baseline I/O suite is designed under the assumption that simulations or analyses occur over multiple time steps, with alternating phases of computation and I/O. This mirrors the conventional patterns in physics simulations, where extensive numerical computation occurs across many time steps to simulate a physical phenomenon. It is common practice to record the state of the simulation in memory for the purposes of monitoring its progress or providing a checkpoint to eliminate the possibility of failure.

No actual computation is carried out in the baseline I/O suite; instead, the `sleep()` function is employed to simulate computation time. The data generated in the write benchmarks is randomized, with the current time serving as a seed. The

read kernel then utilizes the data generated in the write kernel and emulates the data analysis time through the use of the `sleep()` function. The duration of the emulated computation time can be specified in the configuration file.

The baseline I/O suite generates a CSV output file (see Figure 3.7 for an example). This file includes details about the tested operation, such as whether it was a read or write, the number of MPI ranks used, and whether metadata, collective mode (described in Section 3.2.2) or subfiling VFD (described in Section 3.2.4) were employed. A detailed examination of the metrics provided is warranted, as the nature of the data produced by this module had a significant impact on the output of the h5bench extension. This is discussed in further detail in Chapter 5.

```
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
```

Figure 3.7.: Example output csv file for baseline I/O kernels.

Additionally, the CSV file presents numerical characterisations of the total time spent on simulated computations and the total size of written/read data. It also provides raw time, defined as the time in seconds spent directly on the H5Dwrite and H5Dread calls, and raw rate, calculated as the total amount of written/read data divided by raw time. The metadata time metric provides information about the duration of metadata operations in seconds. This includes operations related to opening and closing a dataset or group, but excludes operations related to opening and closing a file containing a dataset, as well as creating or closing a data space when writing or reading a dataset.

The observed time and observed rate are key metrics. Observed time represents the total time in seconds to complete the benchmark. However, the observed rate is not simply the total write/read size divided by the observed time. Instead, it is calculated by dividing the total write/read size by the difference between the observed time and the total compute time. It seems reasonable to posit that this approach was taken by the benchmark creators in order to display only the rate associated with I/O. However, no precise information has been found regarding this hypothesis. Additionally, it is important to note that the data presented represent values that are specific to MPI rank 0 and should be interpreted accordingly.

3.3.2. Running h5bench

To understand the operational principles of h5bench and how to use it, it's helpful to examine its workflow, as shown in Figure 3.8. The program receives a configuration file as input, which determines which benchmark to run and prepares the environment. This step is especially important when using VOL connectors that

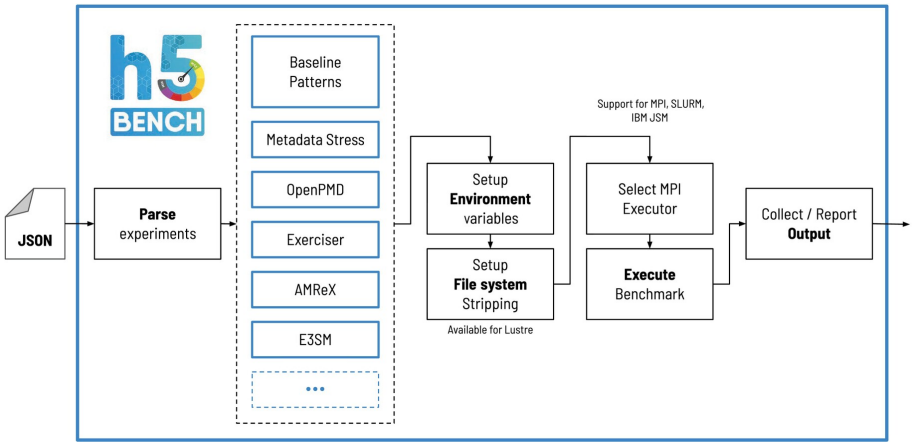

Figure 3.8.: The workflow of h5bench suite.

require specific environment variables. In the event that the measurements are conducted on a Lustre system and the configuration file contains system parameters, the benchmark will set the file system striping. Subsequently, an MPI launcher, such as `srun`, `mpiexec` or `mpirun`, is selected and the benchmark is executed. The result of the benchmark's work is then collected and stored in the appropriate files and directories.

The sample JSON configuration file for h5bench is shown in Figure 3.9 and includes five main sections: mpi, vol, file-system, directory, and benchmarks. Each property must be defined, even if its body is empty. The configuration allows users to specify the MPI launcher and the desired number of MPI ranks to execute. In the event that a more complex configuration is required, or precise control over job parameters is necessary, users are able to define a "configuration" property that h5bench will utilise when executing experiments with the provided "command" property. In the event that a configuration parameter is specified, h5bench will disregard the "ranks" property.

As previously stated, h5bench is compatible with HDF5 VOL connectors. Nevertheless, as not all benchmarks in h5bench are compatible with VOL connectors, it is necessary for users to incorporate the pertinent information into the configuration file in order to regulate the VOL configuration at runtime. In order to achieve this, it is necessary to specify absolute paths to the requisite libraries by utilising the "library" property. Furthermore, the path to the VOL connector must be specified and configured within the "connector" property. The following example illustrates the configuration of the HDF5 asynchronous VOL connector.

h5bench generates a dedicated catalogue for each workflow, wherein all generated files and logs are stored. It is possible to apply additional options, such as data striping for Lustre, to this directory if the relevant configuration is completed. The "file-system" property may be employed by users to configure specific file system options. At present, this property may be employed for Lustre in order to define the number and size of striping for a catalogue that stores data generated by h5bench.

The "benchmarks" property allows users to specify which benchmarks h5bench should run, their order and configuration options. For each h5bench run, the user needs to specify the appropriate configuration. In the event that multiple benchmarks and configurations are indicated within the "benchmarks" section, they will be executed in a sequential order.

In order to execute the h5bench program, the following command must be entered:

```
h5bench configuration.json
```

```
{
    "mpi": {
        "command": "srun",
        "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_TIME_STEP": "1 s",
                "NUM_DIMS": "1",
                "DIM_1": "4194304",
                "DIM_2": "1",
                "DIM_3": "1",
                "CSV_FILE": "output.csv",
                "MODE": "SYNC",
            }
        }
    ]
}
```

Figure 3.9.: Example h5bench configuration file.

3.4. DLIO

The Deep Learning I/O (DLIO) Benchmark [28] is the leading benchmark for AI workloads [12], developed to replicate the I/O patterns and behaviors typical of deep learning applications. It is also part of MLPerf Storage [9]. These factors led to my decision to use DLIO to extract and analyze I/O patterns from AI workloads. DLIO is delivered as an executable and can be adapted to various I/O models through a customizable interface. Its modular architecture allows for the integration of different data formats, datasets, and configuration parameters using modern design patterns, creating a transparent and extensible framework. According to the developers, DLIO's design goals can be summarized into three key objectives: fidelity, configurability, and extensibility. Fidelity refers to accurately replicating I/O behavior, configurability allows simulation of a wide range of usage scenarios, and extensibility enables the modification of the benchmark by incorporating custom data loaders or data generation algorithms.

DLIO is implemented in Python 3.6 and utilizes existing I/O interfaces for various data formats, including TFRecord [78], HDF5 (which I will use in my extension), CSV, NPZ [64] etc. It uses the `sleep()` function to simulate the overlap between computation and I/O operations.

3.4.1. Configuration

DLIO can be configured to support different data loaders, data formats, dataset structures, and training parameters, all through a YAML file.

DLIO's configuration has six main components: workflow, dataset, reader, train, evaluation, and checkpoint. The workflow section specifies which stage of the machine learning process to emulate (data generation, training, or evaluation), along with options for checkpointing and profiling. The dataset covers data setup for the stage specified in the workflow, which includes sample size variation, number of files for training and evaluation, samples per file, chunking settings for HDF5, and so on. The reader covers the batch size, the number of read threads needed to load the data, the transfer size, the shuffle size, and the preprocessing time. In train, the epochs, training steps, and computation time, including its variation, are configured. Evaluation covers the time for the evaluation and the epochs between evaluations. Finally, checkpoint handles the configuration related to the checkpointing. These configurations are integrated into our h5bench extension except the checkpointing and will be discussed in more detail in Chapter 8.

3.4.2. Presets with ML models emulation

DLIO Benchmark provides several ready-to-use configurations based on real-world deep learning applications, such as CosmoFlow or ResNet, available in the DLIO code repository. [6] I have compiled some notable workloads listed in the DLIO

examples in Table 3.2 to give an overview of the configuration that I use in my experiments and how it differs from the configurations of the other AI applications:

Benchmark	CosmoFlow	BERT	UNET3D
backend	tensorflow	tensorflow	pytorch
record-length	2828486	2500	146600628
num-files-train	524288	500	168
num-samples-per-file	1	313532	1
batch-size	1	48	7
read-threads	4	1	4
epochs	5	1	5

Table 3.2.: Configurations of various deep learning kernels in DLIO

In Table 3.2 I only include parameters that play a significant role in the runtime duration of AI applications and have different values between applications. The DLIO Benchmark is provided with a summary of the main components that differentiate the AI workloads from one application to another, which are mainly driven by the component in the dataset and the reader sections in the configuration.

3.5. Profiling tools

In high-performance computing, understanding I/O patterns is crucial for improving application performance and efficiently using computing resources. To achieve this, specialized tools are often used to collect and analyze these patterns. Among the tools commonly used for this purpose are Darshan and DXT, which have become an integral part of the performance analysis workflow. These tools provide invaluable data on I/O behaviour, enabling researchers and engineers to identify inefficient processes, optimise system performance and make informed decisions about system configuration. However, when it comes to DLIO Benchmark, another profiling tool worth mentioning is DFTracer, which comes with the benchmark and is recommended by developers for a detailed study of the generated I/O patterns.

3.5.1. Darshan

Darshan [72] is an I/O characterization tool developed at Argonne National Laboratory that features transparent collection and analysis of I/O access patterns in high-performance computing applications. By providing critical insights into application I/O behavior, Darshan helps improve performance and understand I/O patterns.

Darshan is designed to operate with minimal overhead, thus avoiding interference with the underlying computational processes of applications. It provides an accurate profile of I/O workloads in petascale systems, detailing the patterns of read,

write, and metadata operations performed by applications. Darshan is capable of distinguishing between different access types, particularly MPI and POSIX-based accesses, as well as key HDF5 and PNetCDF [31] functions. It provides a comprehensive view of both intra-file and inter-file access patterns.

Darshan is a transparent user interface that exhibits robust scaling properties, which are crucial for petascale functionality. Implemented as a set of user-space libraries, Darshan integrates easily with applications at the linking stage, replacing standard I/O calls with its own library calls [18]. By intercepting these function calls, Darshan collects comprehensive information about the number of operations (such as open, read, write, stat, etc.), data types, hint usage, alignment, sequencing, access sizes, and timing parameters, storing it in file records for each process. The data collection is conducted independently for each process, with a constrained amount of memory, and does not entail any communication or storage operations during the execution of the application. At the conclusion of the application, the results are aggregated and compressed through the utilisation of MPI reduction operations, collective I/O, and zlib parallel compression. This approach serves to minimise the log file size and reduce overhead, even in the context of large-scale computations. This integration does not necessitate alterations to the source code of the application, thereby streamlining the process for users.

The Darshan command-line component comprises tools for parsing and analysing log files generated by the runtime library. To illustrate, the darshan-job-summary utility furnishes a summary of the application's I/O behaviour in the form of a PDF file (see Figure 3.10). Such summaries may highlight noteworthy details, including discrepancies between the number of MPI-IO collective calls and the number of POSIX calls to MPI-IO records and POSIX record calls, total access sizes, and the distribution of I/O operations over time.

3.5.2. DXT

Darshan eXtended Tracing (DXT) [88] traces application I/O operations, enabling the exploration of diverse workload behaviors. Engineered as a comprehensive I/O profiling instrument, DXT is compatible with the majority of underlying file systems. By default, DXT remains inactive and can be activated by setting the `DXT_ENABLE_IO_TRACE` environment variable to 1. A comprehensive experimental investigation of the overhead introduced by DXT in different I/O access models revealed that the overhead is less than 1%.

DXT comprises a tool that enables the parsing, analysis and visualisation of logs in an offline environment (see example in Figure 3.11). The aforementioned offline analysis is conducted on the specific system or node on which the job was executed, thereby mitigating any potential impact on the performance of the application. DXT is capable of intercepting calls to the most common I/O APIs, including POSIX and MPI-IO, thereby facilitating comprehensive analysis. The analysis tool offers a range of features that facilitate the extraction of valuable insights into the patterns of application I/O access and the relative performance of these operations in relation

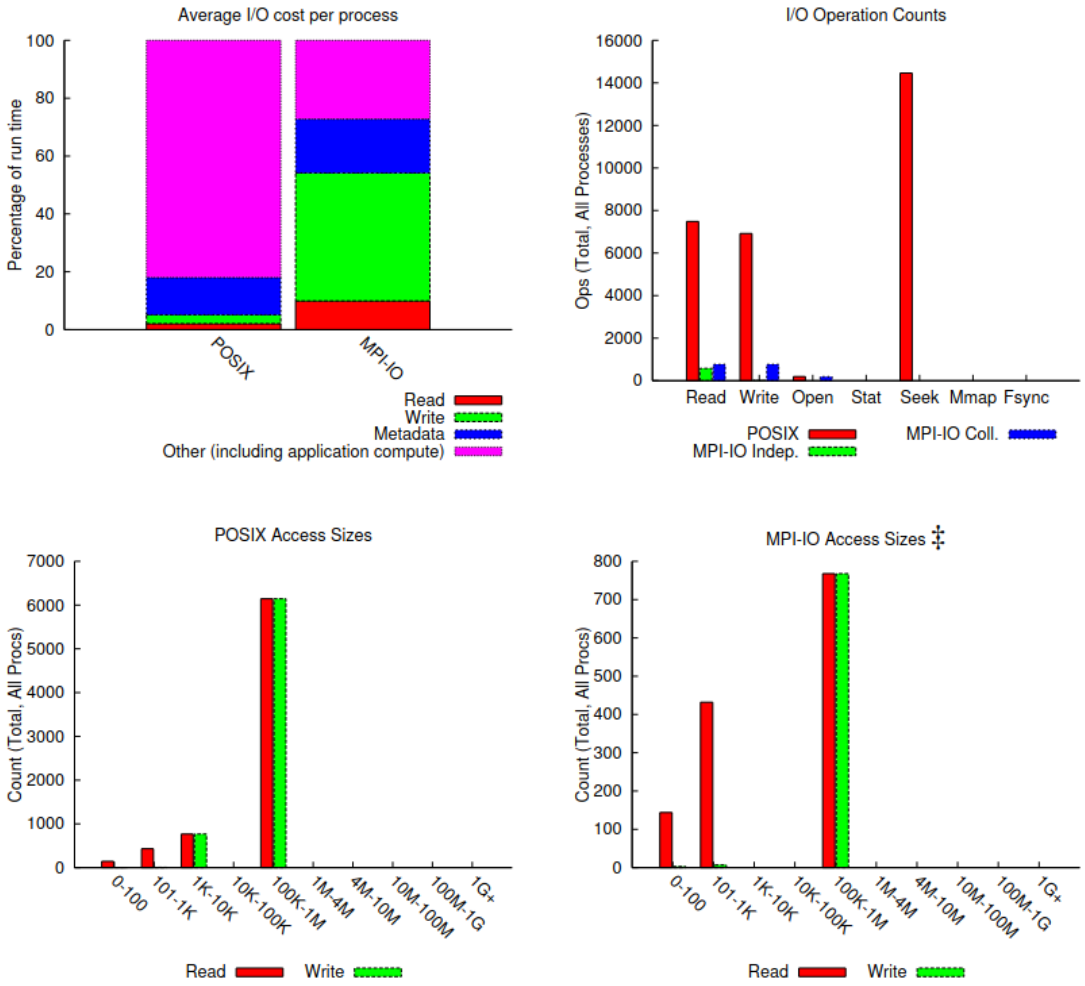

Figure 3.10.: Example of data from the Darshan pdf report. [73]

to the underlying file system. Such features include the capacity to detect non-contiguous I/O, analyse data distribution, generate reports on I/O bandwidth, and identify outliers.

3.5.3. DFTracer

DFTracer [26] (formerly DLIO Profiler until June 2024) is an open-source profiler that has been endorsed by the developers of DLIO Benchmark for the analysis of simulations conducted by the benchmark. DFTracer's main advantage is its ability to perform both application-level and low-level I/O profiling, unlike other tools such as Darshan [72], which are limited to low-level profiling. This dual capability is particularly advantageous for the analysis of complex workloads involving deep learning.

One of the most notable characteristics of DFTracer is its straightforward instal-

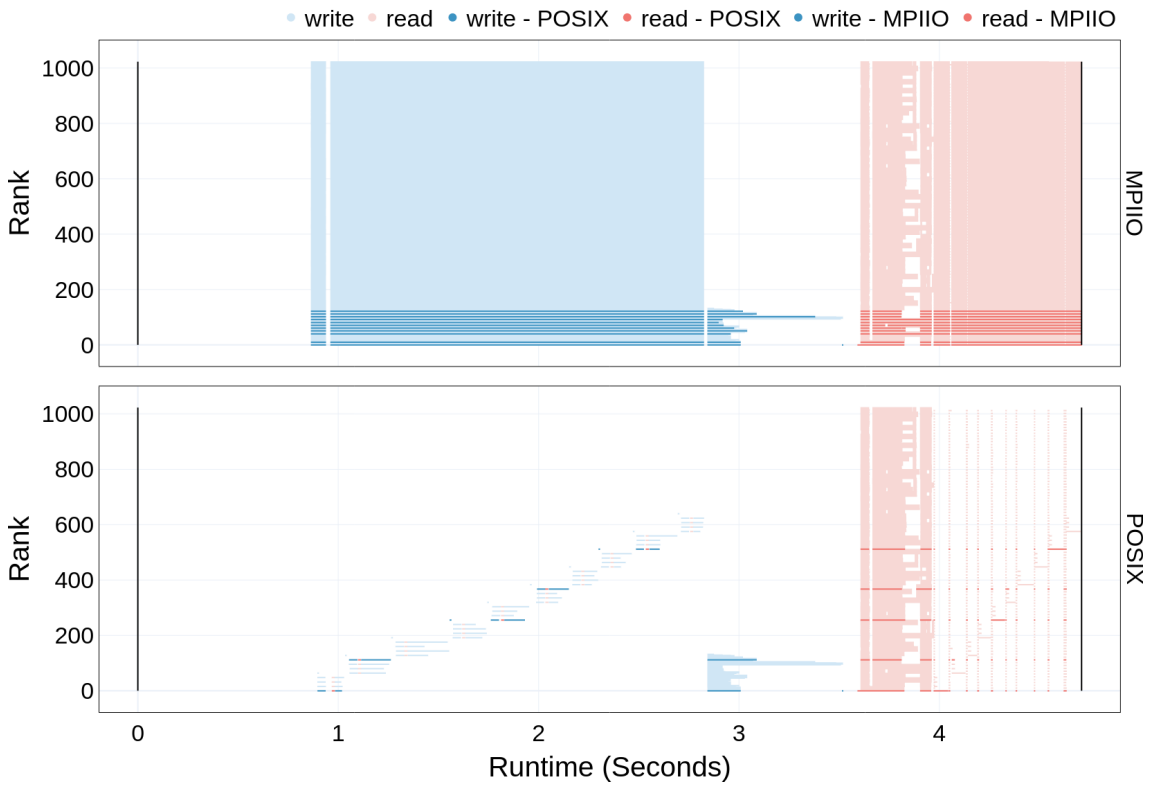

Figure 3.11.: Example of a diagram obtained using DXT. [30]

lation process, as it is readily available as a Python module. In the context of the DLIO Benchmark, DFTracer is invoked for all profiled functions, with the objective of collecting comprehensive data including the function name, the time at which the function was initiated, the duration of the function's execution, the thread within which the function was called, and the arguments that were passed to the function. This data is stored in a .pfw file, which is a JSON-formatted list of objects that can be decoded and analyzed post-execution using tools such as Google's PerfettoUI [56], an open source stack for performance and trace analysis tools.

4. Methodology

This section describes the methods I used to extract I/O patterns from the DLIO Benchmark. The tools used, as described in Chapter 3, include Log VFD, DFTracer, and the combination of Darshan and DXT. To obtain a comprehensive view of I/O performance in machine learning applications, I cross-referenced profiler results with framework documentation and the observed behavior.

Additionally, I analyze the potential design of the h5bench implementation for the extension, based on the extracted I/O pattern information. Finally, I discuss possible methods to compare the equivalency of the DLIO Benchmark and my extension in providing performance insights for the systems they run on.

4.1. DLIO I/O patterns extraction

While extracting I/O patterns from the DLIO Benchmark, I analyzed nearly all the available tuning parameters, which fall into two main groups:

- Dataset settings representing sample sizes, number of samples per file, number of files used, etc.
- Training and evaluation process settings, the main ones responsible for multi-processing settings for data preparation, the order in which the data will be read, number of epochs, batch size and emulated computation time.

Additionally, the latest version of DLIO Benchmark (v1.1) offers the option to select the machine learning library utilized for training and evaluation simulation, including PyTorch and TensorFlow. Given the potential impact of the selected library on I/O patterns, an analysis was conducted to evaluate the effect of different configurations using distinct libraries. The following sections will provide a more detailed account of the extraction of I/O patterns using different tools.

4.1.1. Early Hurdles

In my initial attempt to extract I/O patterns, I used the Darshan and DXT profilers, which are frequently employed for this purpose. As the developers have indicated, Darshan, in addition to providing information about POSIX and MPIO calls, also supports HDF5 [72]. This further reinforced my conviction that this tool should be employed. However, after running darshan-runtime with DLIO Benchmark for the first time, I was getting

```
symbol lookup error:  libdarshan.so:  undefined symbol:  H5FDperform_init
symbol lookup error:  libdarshan.so:  undefined symbol:  H5Eset_auto2
```

errors depending on the version. After discussing the error with the Darshan developer [29], the problem was solved by kludge: adding the path to the HDF5 library to the `LD_PRELOAD` variable, which specifies which libraries should be preloaded when the program is launched.

Running profiling for DLIO Benchmark again I found that it would not generate a section containing HDF5 call data, only a POSIX section. According to my assumption, absence of HDF5 section in Darshan log is due to the fact that Darshan was designed for low level I/O and not application level profiling and thus could not catch the corresponding calls from the machine learning libraries used by DLIO Benchmark that read from HDF5 files.

Following an unsuccessful attempt to obtain data on HDF5 calls, an analysis was conducted on POSIX call information for HDF5 files, which were used in the training and evaluation process. This included an examination of the number of times the file was opened, the number of read operations performed on it, and the total and maximum number of bytes that were read. Furthermore, the utilisation of DXT facilitated the acquisition of data pertaining to the number of instances in which data of varying sizes were accessed. This data was categorised into ranges 0-100 bytes, 100 bytes to 1 Kilobyte, 1 Kilobyte to 10 Kilobytes, and so on. Additionally, the timestamp and duration of the read operations were recorded in seconds. Although this information provided insight into the DLIO Benchmark's interaction with HDF5 files, it remained limited and incomplete: the data was insufficient to form a comprehensive understanding of the interaction. Moreover, it would have been impossible to further design the extension based on the obtained data due to the complex patterns produced by HDF5 calls, so after the unsuccessful experience with Darshan and DXT I switched to other ways of benchmark profiling.

4.1.2. Log VFD

To capture only the HDF5 calls made during DLIO Benchmark runs, I used the Log VFD, which can be enabled by setting the `HDF5_DRIVER` environment variable to "log" in the command line. For further analysis of the obtained data (which will take place in Chapter 6) it is necessary to consider in detail what types of memory exist and what operations can be performed on the data in terms of HDF5. All memory types corresponding to the different sections of the HDF5 file are described in Table 4.1. Operations on these sections include reading, writing, and allocating of memory. It should be noted that there are other operations, but they are not relevant to the interpretation of the data obtained and therefore their description can be omitted.

In addition to identifying the relationship between the various DLIO Benchmark parameters and the resulting I/O patterns, it is also necessary to identify whether

Memory type	Description
H5FD_MEM_SUPER	Management structure, including the signature identifying the file as an HDF5 file and the address of the first block of the file, which helps locate other data structures.
H5FD_MEM_DEFAULT	Indicates that the memory type has not yet been set. Can also be the datatype set in a larger allocation that will be suballocated by the library.
H5FD_MEM_OHDR	An object header that provides dataspace, datatype, and other information that the library uses to speed up access to the object's data.
H5FD_MEM_LHEAP	A local heap is a collection of small pieces of data related to metadata, such as small attribute names and values.
H5FD_MEM_GHEAP	A global heap containing various information common to all datasets.
H5FD_MEM_BTREE	Contains a B-Tree search tree that stores various addresses in a file as the last level leaves to quickly search for objects in the file.
H5FD_MEM_DRAW	Raw data such as the contents of datasets.

Table 4.1.: Description of different HDF5 memory types and file partitions

there is a difference when using the PyTorch and TensorFlow libraries. In particular, the pattern analysis should demonstrate whether there is a discrepancy in the implementation of workers for different libraries that allow parallel sample reading and model training or estimation with respect to HDF5 calls.

4.1.3. DFTracer

To gain a deeper understanding of the training and evaluation simulation of deep learning models in DLIO Benchmark, I also employed the DFTracer profiler. The principal distinction between this and Log VFD is that it records not merely the HDF5 library calls, but all functions invoked within the context of DLIO Benchmark. Additionally, it logs the POSIX calls responsible for generating these functions, including file operations like opening, closing, reading, and writing. This can provide deeper insights into the architecture and implementation of the benchmark, which will undoubtedly influence its further development. An additional benefit of DFTracer is that the .pfw files it generates can be plotted to offer a comprehensive illustration of the benchmark's operational mechanisms and its access patterns to HDF5 files and their constituent parts. As with Log VFD, this profiler will allow for the observation of any discrepancies in the implementation of multiprocessing by workers when utilising disparate ML libraries.

4.2. Implementation design

Once the I/O patterns - both for accessing HDF5 files and the data within them - are obtained, they can be used to develop an extension for h5bench. As h5bench is designed with future extensibility in mind, it already possesses a sufficient degree of flexibility, as well as clear instructions for implementing new features. This extension would consist of a separate module containing one main C file and, optionally, several auxiliary ones. These files would emulate the process of training and evaluating deep learning models based on the extracted patterns. Following

the construction of h5bench with the CMake utility, the module files will undergo compilation and subsequently be made available in the form of an executable. This executable file will be invoked by a Python script, representing the primary file of the benchmarking suite. The implementation, therefore, involves the creation of a module that reproduces the identified patterns and measures performance with a variety of metrics. A more detailed account of this implementation is presented in Chapter 5.

The extension does not need to import machine learning libraries, which would otherwise result in a large call stack. Therefore, the extension is expected to run faster by avoiding unnecessary overhead. Furthermore, the extension can be adapted to include support for HDF5-specific features, such as various VOL Connectors and VFDs not present in the DLIO Benchmark. This would make the extension a unique tool for researchers, offering valuable insights into the performance of AI workloads that utilize the HDF5 format.

4.3. Implementation validation

Once the extension for h5bench has been developed, it must be evaluated against the DLIO Benchmark. The two principal aspects that will be subjected to examination are as follows. The first test will assess the equivalence of access patterns to HDF5 files using the same extensions and DLIO Benchmark settings. If the settings are identical, the patterns are anticipated to be identical. The Log VFD output will be used to test the feasibility of this criterion. However, since Log VFD does not provide runtime-related metrics, the second test will compare the execution times of both programs, focusing on the time spent simulating training and evaluation. This criterion is especially important, as the main goal of the extension is to establish a faster benchmark for evaluating I/O and scalability in AI workloads. In contrast, the DLIO Benchmark requires installing and executing the entire AI pipeline, including PyTorch and TensorFlow. To meet the second criterion, the extension must demonstrate results proportional to those of the DLIO Benchmark under the same configurations, but with a shorter runtime.

5. Implementation

This section provides a detailed account of key implementation aspects. In Section 5.1, I describe the general workflow of the benchmark and highlight some of the most important parameters (a complete list is available in Appendix B). Section 5.2 covers the implementation of multiprocessing to manage workers. Next, Section 5.3 will detail the main function of the benchmark, which is responsible for generating the I/O pattern. Finally, Section 5.4 discusses the performance metrics generated by the benchmark.

5.1. Workflow

The workflow of the extension mirrors the basic structure of most machine learning applications, as shown in Figure 5.1.

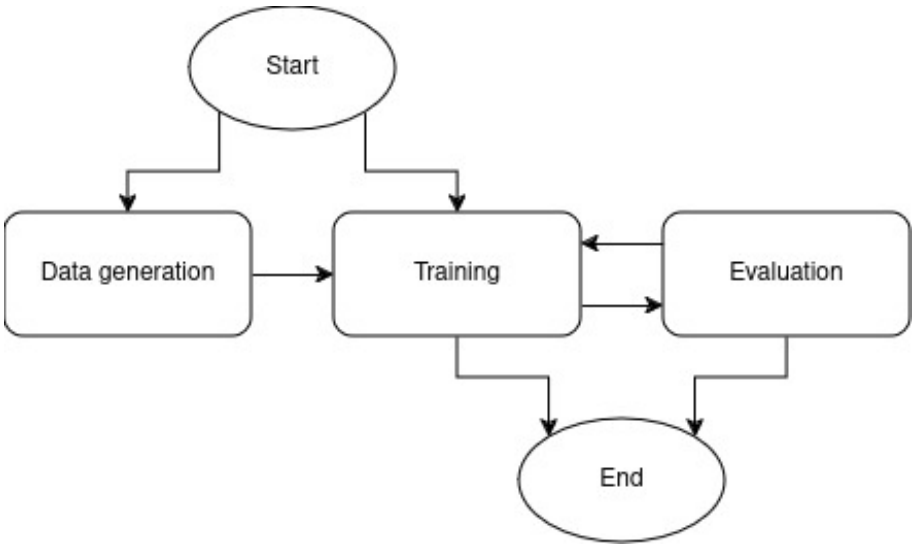

Figure 5.1.: A sketch of the h5bench extension workflow.

If the user does not have a suitable dataset for emulating the training and evaluation processes, one can be generated. In the process of generating the dataset, two directories are created (named "train" and "valid" by default, although the names can be modified using the *train-data-folder* and *valid-data-folder* settings). These directories contain HDF5 files. The number of files is determined by the settings designated as *num-files-train* and *num-files-eval*, respectively. All MPI ranks are

engaged in the generation of files, with each thread assuming responsibility for the creation of a designated number of HDF5 files, except in instances where a connected Subfiling VFD is in use. In such cases, all files are generated collectively in a sequential manner. Each generated file comprises two datasets: records and labels (the latter of which can be configured via the *records-dataset-name* and *labels-dataset-name* settings, respectively). The labels dataset is populated with 64-bit zeros, corresponding to the number of samples per file. Notably, this dataset is not used in the training or evaluation simulations. It has been incorporated solely for the purpose of emulating the I/O DLIO Benchmark patterns in their entirety. The records dataset is a three-dimensional dataset comprising two-dimensional samples. The size of each sample is given by $\sqrt{\textit{record-length}} \times \sqrt{\textit{record-length}}$. The number of samples in the file corresponding to the third dimension is defined by the parameter *num-samples-per-file*. A random number buffer of up to 2 GB is generated for the dataset. In the case of a records dataset with a size of less than 2GB, the requisite number of values will be generated. Each random number is represented by an 8-bit integer value. Upon completion of the generation process for all HDF5 files, the training simulation process commences.

The training simulation involves sequentially opening and reading HDF5 files and their samples by distinct MPI ranks, repeated for the specified number of epochs. The sequence of accessing files and samples is randomized via the *shuffle* parameter, and regenerated with each epoch. Following the reading of each sample, computations are simulated by the `sleep()` function, the duration of which is configured by the *preprocess-time* parameter. In some instances, it may be preferable for the emulated running time to vary, in order to more closely align with the actual computational process. In such instances, the *preprocess-time-stdev* parameter may be employed to establish the standard deviation associated with the final simulation time of preprocessing samples. Once the sample batch has been read (the size of which is also configurable via the *batch-size* parameter), the model training simulation is initiated. This is also implemented through the `sleep()` function and configured by the *computation-time* and *computation-time-stdev* parameters. In the event that the user does not require the simulation of model training on the entirety of the data set, the total number of sample reads for all MPI ranks can be limited through the utilisation of the parameter *total-training-steps*.

In contrast to the training simulation, the model evaluation simulation is not required to occur at each epoch. Its configuration is determined by the *epochs-between-evals* parameter, which stipulates the number of epochs over which the estimation process will be simulated. This process, like the training process, involves the sequential reading of batches of samples from different HDF5 files in a specific sequence, with the preprocessing occurring after each reading. The sole distinction between the two lies in the utilisation of the *eval-time* and *eval-time-stdev* parameters, which are employed to define the simulation computation time.

5.2. Multiprocessing

A key feature of the implementation is multiprocessing, which enables parallel reading, preparation, and packing of samples into batches before training or evaluation simulation. Consistent with the emulated PyTorch and TensorFlow frameworks used in DLIO Benchmark, the default Unix `fork()` method is utilised for multiprocessing. [3] Before the benchmark starts, if the *read-threads* parameter is set to a non-zero value, a specified number of new processes are created independently for reading data during both the training and evaluation phases for each MPI rank. Thus, if the benchmark utilises eight MPI ranks and the read-threads parameter is set to four, the total number of processes created will be 32. Communication pipes are established between the child processes and the main process. Subsequently, each worker receives sequences of samples to read and performs an independent simulation of the reading and preprocessing. After reading and preparing the sample batches, the worker signals the parent process, which commands it to prepare the next batch. Meanwhile, the worker itself performs the computation simulation. The distribution of tasks pertaining to the reading of batches between the workers is conducted in accordance with the principles of the thread pool design pattern. It is worth noting that the case in which the *read-threads* parameter is equal to zero is particularly noteworthy. In such cases, no new processes will be created, with all operations pertaining to reading, data preparation and computation simulation instead being performed solely by the parent process - various MPI ranks.

5.3. read_sample()

The fundamental aspect of the implementation is the process of reading a specific sample from a designated file. The `read_sample()` function is responsible for all interactions with HDF5 files, emulating I/O patterns. The code of the function is presented in Listing 1. The function takes a file path string, representing the path to the HDF5 file, and an integer specifying which sample in the file to read. For clarity, the function can be divided into three parts. Initialisation of the resources is the first stage of the process. This involves opening the relevant HDF5 file and dataset, preparing the filespace and memoryspace, and allocating memory for the data to be read. Subsequently, the data is read directly using the `H5Dread()` function. After that the memory allocated for reading is freed, and in the final stage the resources associated with the opening of the HDF5 file are released. The function returns two variables: `read_time_out = t4 - t3`, representing the execution time of the `H5Dread()` function and `metadata_time_out = (t2 - t1) + (t6 - t5)`, which contains the time spent on preparing and closing the HDF5 file, namely on the `H5Fopen`, `H5Dopen`, `H5Dget_space`, `H5Screate`, `H5Sselect_hyperslab`, `H5Sclose`, `H5Dclose`, and `H5Fclose` functions. These two values that are involved in most of the system performance summary calculations and called raw read time and metadata time, as detailed in Section 5.4.

```
void read_sample(
    file_path, sample, *metadata_time_out, *read_time_out
) {
    /* (1) Initialization */
    offset = [sample, 0, 0]
    count = [1, DIM, DIM]

    t1 = get_current_time_in_microseconds()

    file_id = H5Fopen(file_path, H5F_ACC_RDONLY, FAPL)
    dataset_id = H5Dopen(file_id, RECORDS_DATASET_NAME, DAPL)
    filespace = H5Dget_space(dataset_id)
    memspace = H5Screate_simple(3, count, NULL)
    H5Sselect_hyperslab(
        filespace, H5S_SELECT_SET, offset, NULL, count, NULL
    )

    t2 = get_current_time_in_microseconds()

    data = allocate_memory(DIM * DIM * size_of(uint8_t))

    /* (2) Collecting read information */
    t3 = get_current_time_in_microseconds()
    H5Dread(
        dataset_id, H5T_STD_U8LE, memspace, filespace, DXPL, data
    )
    t4 = get_current_time_in_microseconds()

    /* (3) Releasing resources */
    free_memory(data)

    t5 = get_current_time_in_microseconds()

    H5Sclose(memspace)
    H5Sclose(filespace)
    H5Dclose(dataset_id)
    H5Fclose(file_id)

    t6 = get_current_time_in_microseconds()
}
```

Code 1: Source code of the `read_sample()` function responsible for reading data.

5.4. Understanding the output

This section provides a detailed description of the performance metrics generated by the benchmark, along with an explanation of the calculations used to derive them. Upon completing the benchmarking process, a CSV report is generated that contains the average performance for each MPI rank. Additionally, reports for each individual MPI rank can be generated by enabling the *output-ranks-data* parameter. Table 5.1 shows an example of the report in tabular form, which will be explained in detail.

metric	value	unit
operation	dlio	
ranks	16	
read threads	0	
subfiling	NO	
chunking	NO	
collective meta	YES	
collective data	YES	
train total size	63.000	GB
train size per rank	3.938	GB
train emulated compute time per epoch	2.907, 2.907, 2.907	s, s, s
train emulated compute time	8.721	s
train metadata time per epoch	0.559, 0.502, 0.434	s, s, s
train metadata time	1.495	s
train raw read time per epoch	4.205, 3.835, 3.767	s, s, s
train total raw read time	11.807	s
train raw read rate per epoch	958.749, 1.026, 1.045	MB/s, GB/s, GB/s
train avg raw read rate	1.003	GB/s
train observed time per epoch	8.709, 8.125, 7.789	s, s, s
train observed time	24.623	s
train observed rate per epoch	694.948, 772.636, 825.826	MB/s, MB/s, MB/s
train avg observed rate	764.47	MB/s
train throughput samples per second per epoch	7.234, 7.753, 8.088	samples/s, samples/s, samples/s
train throughput avg samples per second	7.692	samples/s
train throughput stdev samples per second	0.351	samples/s
train io avg	492.267	MB/s
train io stdev	22.485	MB/s
eval total size	16.000	GB
eval size per rank	1.000	GB
eval emulated compute time per epoch	2.584, 2.584, 2.584	s, s, s
eval emulated compute time	7.752	s
eval metadata time per epoch	0.214, 0.151, 0.162	s, s, s
eval metadata time	0.527	s
eval raw read time per epoch	0.925, 0.913, 0.875	s, s, s
eval total raw read time	2.713	s
eval raw read rate per epoch	1.080, 1.095, 1.143	GB/s, GB/s, GB/s
eval avg raw read rate	1.106	GB/s
eval observed time per epoch	4.120, 3.904, 3.881	s, s, s
eval observed time	11.905	s
eval observed rate per epoch	666.574, 775.895, 789.646	MB/s, MB/s, MB/s
eval avg observed rate	744.038	MB/s
eval throughput samples per second per epoch	3.883, 4.099, 4.123	samples/s, samples/s, samples/s
eval throughput avg samples per second	4.035	samples/s
eval throughput stdev samples per second	0.108	samples/s
eval io avg	258.24	MB/s
eval io stdev	6.907	MB/s

Table 5.1.: Example of csv file with benchmark results.

The first seven lines, following the format of other h5bench kernels, display the benchmark's name along with key configurations, such as the number of MPI ranks, the number of read threads, and whether Subfiling VFD, chunking, or collective MPI mode were used.Next, the report shows the total amount of data used to emulate the training process, along with the data volume processed by each MPI rank. It then provides the total emulated computation time for each epoch (in this example, three epochs) and for all epochs combined. The next four lines present data on the time required to process metadata and raw reads for each epoch and for all epochs combined. This information was previously discussed in Section 5.3. From this data, the raw read rate is calculated as:

$$\text{Raw read rate} = \frac{\text{Amount of data used to emulate the training}}{\text{Raw read time}}$$

In addition to the average raw read rate across all epochs, the observed read time is considered. This represents the total time spent on the entire training simulation process. The observed time is calculated similarly to the raw read rate:

$$\text{Observed read rate} = \frac{\text{Amount of data used to emulate the training}}{\text{Observed read time}}$$

The training process statistics are finalized with the metrics throughput and training IO, calculated using the following formulas:

$$\text{Throughput} = \text{Batch size} \times \frac{\text{Training steps per rank}}{\text{Observed read time}}$$

$$\text{Training IO} = \text{Throughput} \times \text{Record size}$$

These metrics provide insight into the IO throughput, expressed in both samples per second and megabytes per second. All the formulas and counting methods described above are also valid for the model evaluation simulation process.

6. Experiment results

This section presents a description of the experimental setup and a discussion of the I/O patterns obtained using Log VFD and DFTracer. I will then investigate whether the I/O patterns of my implementation are equivalent to the DLIO Benchmark patterns, thus closing the first criterion for a valid solution. Once the runtime of both programs has been compared and it has been established that the difference is constant and that the extension is more efficient, the second criterion will be satisfied, thereby confirming the validity of the implementation. Both criteria were described in Section 4.3. At the end, I will present the results of scaling tests for different extension configurations, fully illustrating all the main features.

6.1. Experimental setup

All research work and measurements were conducted on CLAIX-2023 [85] [82], a computing cluster at RWTH Aachen University. The cluster was utilised with varying numbers of nodes and cores, and with both the Lustre [70] and BeeOND [59] file systems, which were operated via the Slurm batch system [92]. The following versions of the libraries have been used: DLIO Benchmark (v1.1), DFTracer (v0.0.3), h5bench (v1.5), HDF5 (v1.14.0) with build flags

- `-DHDF5_ENABLE_PARALLEL=ON`
- `-DHDF5_ENABLE_THREADSAFE=ON`
- `-DALLOW_UNSUPPORTED=ON`
- `-DHDF5_ENABLE_SUBFILING_VFD=ON`

The Intel MPI environment, version 2021.6.0, was employed in the implementation of the h5bench extension. The experiments were conducted using various MPI launchers, including `mpiexec` and `srun`, given the differing configuration of the CLAIX-23 cluster, which can result in notable performance variations. Experiments with DLIO Benchmark were conducted using Python 3.10.4, PyTorch 2.3.1, and TensorFlow 2.16.2. Simulations were performed with computing resources granted by RWTH Aachen University under project thes1744.

All experiments were ran with at least ten times repetitions to check the volatility of the performance numbers. The list of experiments is as follows:

- **Performance comparison on Lustre and BeeOND** - 4 nodes (32 processes in total) in Lustre & BeeOND with configuration A (presented in Table B.1) and `mpiexec` MPI launcher.
- **Unet3D workload comparison** - 4 nodes (32 processes in total) in BeeOND with configuration B (presented in Table B.1) and `mpiexec` MPI launcher.
- **Read threads performance comparison** - 1 node (8 processes) with configuration C (presented in Table B.1) and `srun` MPI launcher.
- **Scaling test** - 1 node with 1 to 64 numbers of processes in Lustre and BeeOND with configurations D, E, F, G, H (presented in Table B.2) and `srun` MPI launcher

All experimental results obtained can be found in Appendix C.

6.1.1. CLAIX-2023

As mentioned before, all measurements were performed on the CLAIX-2023 cluster located in Aachen, Germany at RWTH Aachen University and was put into operation in 2024. It has a fat-tree topology and consists of 684 nodes, 632 of which contain 2 Intel Xeon 8468 Sapphire Processors with a clock speed of 2.1 GHz and 48 cores each. There from 256 GB to 1024 GB of RAM memory per node. The total theoretical peak performance is around 4 PFLops. A high-performance Lustre-based storage system offers a capacity of 26 petabytes and a bandwidth of 500 GB/s (read and write). InfiniBand technology is used for internal interconnect.

6.1.2. Lustre

One of the available file systems on a cluster is Lustre, a distributed file system typically used for large-scale cluster computing. The Lustre architecture consists of many components, including object storage targets (OSTs), metadata servers (MDSs), and clients. Files in Lustre are distributed across multiple OSTs, which provides high-speed access by distributing read and write operations across multiple storage nodes. Metadata, such as directory structure and file attributes, is managed by the MDS to help you quickly locate and access file data. The separation of metadata and data allows Lustre to efficiently handle a large number of files and operations simultaneously. One of the key advantages of Lustre is its scalability. It can support thousands of clients and petabytes of data while maintaining consistent performance.

The Lustre provided on the cluster is of RAID 0 type [19]. Stripe count 1 is set for file size less than 1 GB, stripe count 4 for file size between 1-64 GB and stripe count 16 for file size larger than 64 GB. The stripe size is 16 MB. It is also necessary to mention that CLAIX-23's Lustre is a production-level shared storage resource where other users' activities in Lustre can affect my experiments.

6.1.3. BeeOND

Another available file system on the cluster is BeeOND (BeeGFS on Demand). 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.

The BeeOND provided on the cluster is also of RAID 0 type with 512k chunk size, four storage targets, and one storage pool. Each node for the BeeOND filesystem has a 1.4 TB SSD. Unlike the Lustre filesystem, BeeOND nodes are used exclusively and are not shared with other users. The impact of this exclusive usage can be seen in Figure 6.6

6.2. Analysis of the extracted I/O patterns

6.2.1. Log VFD

In order to obtain the I/O patterns generated by the DLIO Benchmark, a detailed analysis was conducted on a significant number of Log VFD logs, obtained for a variety of benchmark configurations. In them I found that each read operation corresponds to the following block of records:

```
  0-         8 (         8 bytes) (H5FD_MEM_SUPER) Allocated
  0-         7 (         8 bytes) (H5FD_MEM_SUPER) Read
  8-        16 (         8 bytes) (H5FD_MEM_SUPER) Allocated
  0-        15 (        16 bytes) (H5FD_MEM_SUPER) Read
 16-        96 (        80 bytes) (H5FD_MEM_SUPER) Allocated
 16-        95 (        80 bytes) (H5FD_MEM_SUPER) Read
 96- 268437536 ( 268437440 bytes) (H5FD_MEM_DEFAULT) Allocated
 96-       607 (       512 bytes) (H5FD_MEM_OHDR) Read
680-      1191 (       512 bytes) (H5FD_MEM_LHEAP) Read
136-       679 (       544 bytes) (H5FD_MEM_BTREE) Read
1072-      1399 (       328 bytes) (H5FD_MEM_BTREE) Read
800-      1311 (       512 bytes) (H5FD_MEM_OHDR) Read
2048-  67110911 (  67108864 bytes) (H5FD_MEM_DRAW) Read
```

Given that the entirety of the training and evaluation of models in relation to HDF5 files is essentially limited to the reading of data, it is unsurprising that the

final logs comprised a significant number of such blocks. Let's take a closer look at the records in a block. The initial six records are responsible for memory allocation and subsequent reading of records from the superblock, the HDF5 file header. This header contains critical metadata, including the file signature, version information, root group pointer, base address, file size, and driver information. This ensures that the file is identifiable, self-describing, and accessible across platforms. The next line of code allocates memory for the entire file, excluding the superblock. This is indicated by the byte range, which begins at 96, corresponding to the end of the superblock, as evidenced by the preceding six entries. This Log VFD record block describes the calls that occur when an HDF5 file containing 256 megabytes of data or four samples is opened and the first 64 megabytes of the sample are read. This is why the size of the allocated memory is 268437440 bytes, which corresponds to 256.002 Megabytes. This is the memory required for the data stored in the file, as well as the memory required for some metadata structures. The use of these structures can be seen in the next five lines of the log, which read the metadata of groups and datasets, including their names, as well as information about the data layout stored in B-Tree. The final line of the log describes the direct reading of the data written to the HDF5 file. This confirms that the number of bytes read is 67108864, which is equal to 64 Megabytes.

The seventh line, which is responsible for memory allocation for data and some structures, undergoes changes only in the number of bytes allocated, depending on the specified file size and the corresponding range. Similarly, the last line, which is responsible for direct data reading, also exhibits alterations in the range and the number of bytes read, contingent on the configured sample size and the sample in the file being read. To illustrate, when the second 64 MB sample is read from a 256 MB file comprising four samples, the final line of the log block will appear as follows:

```
67110912- 134219775 (  67108864 bytes) (H5FD_MEM_DRAW) Read
```

It can be observed that the size of the read data remains consistent when compared to the previous line from the aforementioned block, as both instances involve the reading of 64 Megabytes of data. However, the range has undergone a change. The sequence now commences with the value 67110912, which is precisely one byte greater than the ultimate value of the range observed in the preceding block (67110911). The new end range value is equal to the current start value plus the sample size, that is 67110912 + 67108864 - 1 = 134219775. Upon examination of the logs generated by the third and fourth samples, it can be observed that the ranges remain predictable and consistent with the calculated values.

```
134219776- 201328639 (  67108864 bytes) (H5FD_MEM_DRAW) Read
201328640- 268437503 (  67108864 bytes) (H5FD_MEM_DRAW) Read
```

6.2.2. DFTracer

As previously stated, DFTracer offers not only application-level profiling, but also low-level profiling, which enables the drawing of parallels between high-level functions in Python, on which DLIO Benchmark is based, and low-level POSIX operations. Figure 6.1 gives an idea of the kind of POSIX information available in the profiler. These can already be correlated with calls related to HDF5 files using Log VFD logs. An example of the visualised profiler output for a single MPI rank is provided in Figure 6.2.

It displays the entire benchmark workflow related to HDF5 file generation, training and evaluation simulations. The length of each bar represents the execution time of a specific function. Therefore, a longer bar indicates a longer execution time for the corresponding function. The image displays the generation process in the left half. An analysis of the data suggests that the generation of the HDF5 dataset is distributed equally among all MPI ranks.

Figure 6.1.: Sample data available for POSIX operations.

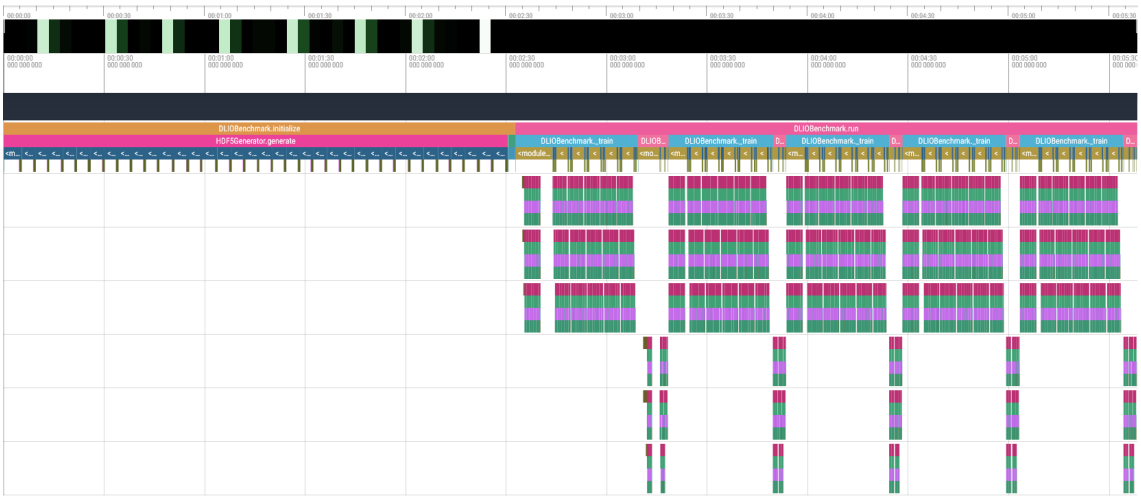

Figure 6.2.: Plot generated on DFTracer received data for configuration with PyTorch and 3 read threads.

No distinctive pattern was identified in the generation of files; they are simply created one by one through the random generation of numbers. The process of emulating the training (top) and evaluation (bottom) of the model is visible on the right side of the image. In the presented configuration, three read threads were utilized, as illustrated in the image, with each thread corresponding to a distinct line. It can be observed that distinct workers are employed to simulate the training and evaluation procedures. The model training and evaluation processes undergo changes throughout the five epochs, yet the same workers are reused when utilising PyTorch. However as shown in Figure 6.3, this is not the case with TensorFlow. It was found that TensorFlow creates new workers with the commencement of each epoch. The responsibility of each worker is limited to the reading of a sample, or a patch comprising multiple samples, from a predefined sequence set by the main thread. Subsequently, the data is conveyed to the main thread, which then initiates the computation simulation process. This is the case for both the training and evaluation processes. Furthermore, the image illustrates that the training simulation process and the model evaluation process have disparate durations. This discrepancy can be attributed to the differing sizes of the training and evaluation datasets, differing by a factor of four.

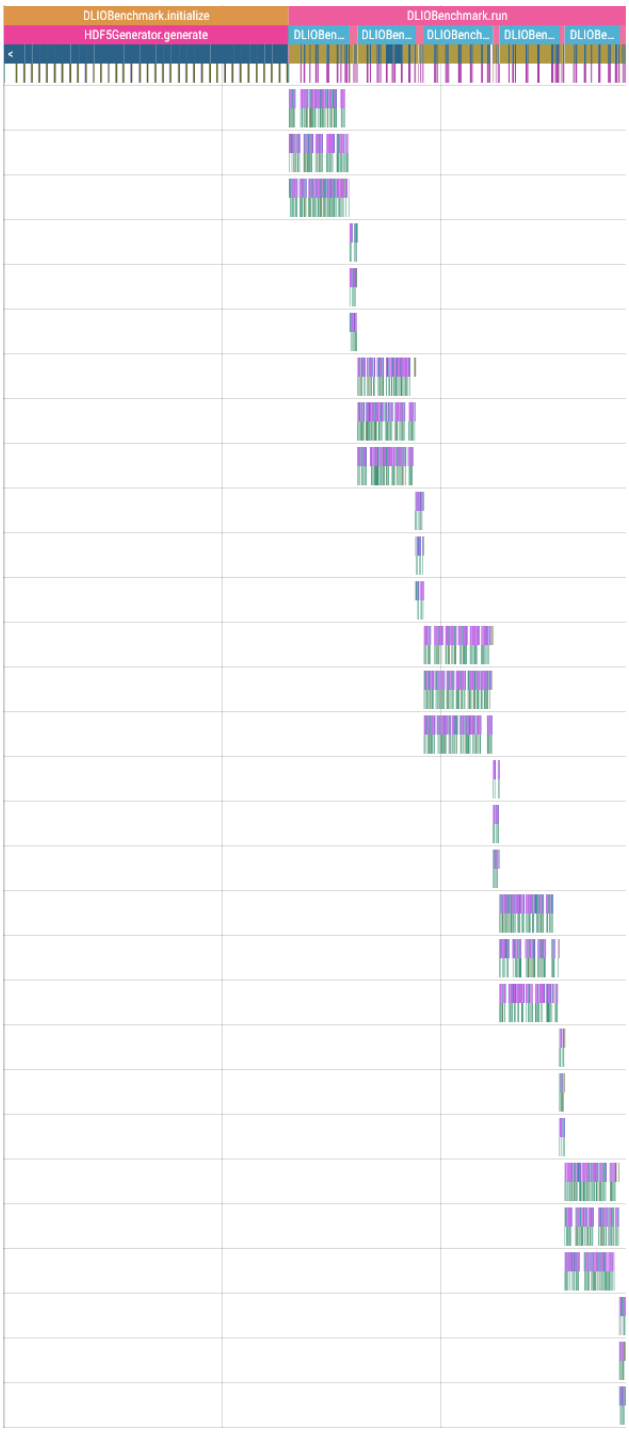

Figure 6.3.: Plot generated on DFTracer received data for configuration with TensorFlow and 3 read threads.

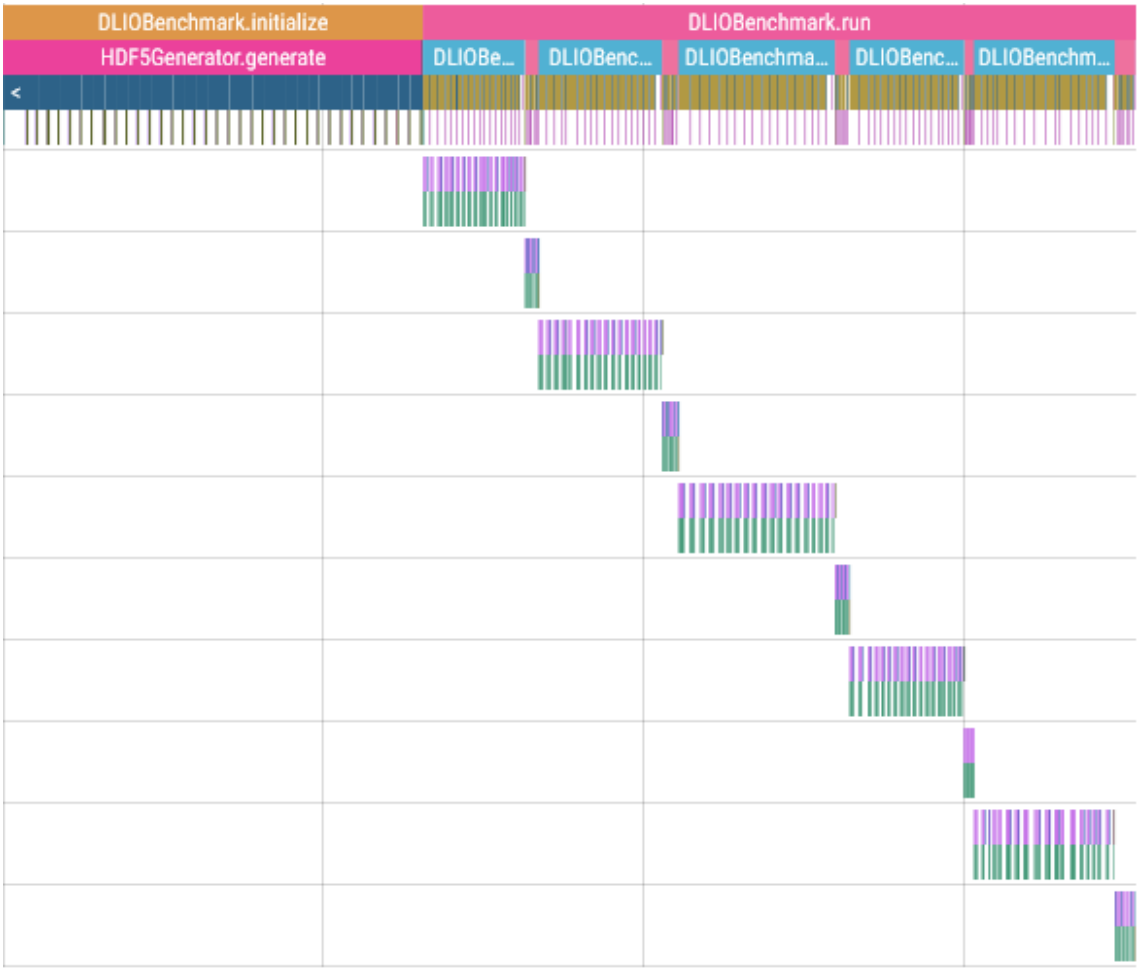

Figure 6.4.: Plot generated on DFTracer received data for configuration with PyTorch and 3 read threads.

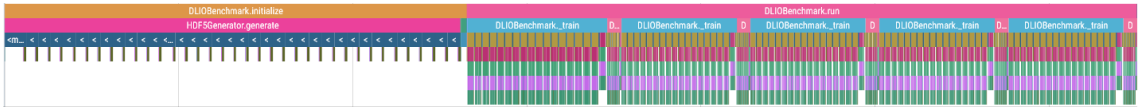

Figure 6.5.: Plot generated on DFTracer received data for configuration with PyTorch and 0 read threads.

Figure 6.4 shows the result of the profiler using all the same configurations but with the number of read threads set to zero. In this case, we can see that by default, TensorFlow still creates a separate thread (worker) for each epoch. This behaviour is described in detail in the DLIO benchmark documentation. [83] If the same configuration with zero read threads is run with PyTorch simulation, it can be seen from Figure 3 that no new threads are actually created, as indicated by the fact that all function calls on the graph occur in a single thread - the parent thread.

6.3. Implementation validation

The next few sections will detail the implementation verification process using Log VFD. In addition, this section will compare the performance metrics, as described in Section 4.3, namely runtime and peak fabric bandwidth between the DLIO benchmark and my extension, achieved using the Lustre and BeeOND parallel file systems.

6.3.1. Log VFD

To validate the implementation, a comparison was conducted between the Log VFD logs of different configurations common to the implemented extension and the DLIO Benchmark. In all cases, identical entries were obtained up to permutation. If the random sample shuffling was disabled, the logs were identical, thereby confirming that the extension successfully emulates the DLIO Benchmark-generated I/O patterns in terms of HDF5 calls. Table 6.1 shows the number of Log VFD records obtained in the experiment with a sample size of 67108864 bytes, the number of files for training and evaluation configured as 128 and 32, respectively. The number of samples per file was 4, and the number of read threads was also 4. The experiment lasted for one epoch, and the number of steps in the training phase was limited to 511. As the logs for the extension and DLIO Benchmark are identical, the table shows the number of log entries common to both programs. Thus, the number of records with metadata and file structure is 639. This number can be deduced with ease by calculating the total number of accesses to HDF5 files during training and evaluation simulations:

- The number of files utilized for training purposes (128) is multiplied by the number of samples within each file (4), resulting in a total of 512. Given that the number of steps in the training phase is constrained to 511, it can be inferred that the HDF5 files were accessed 511 times throughout the training process. Of these, 384 were accessed for reading the first three samples, while 127 were accessed for reading the last one.

- The number of files to be evaluated is 32, and the number of samples in each file is 4. Therefore, the total number of samples to be evaluated is 128. Consequently, the number of file calls is 128, with 32 calls made for each of the first, second, third and fourth samples.

Thus, the total number of accesses to the file is the sum of the accesses during the training and evaluation simulation, namely 511 + 128 = 639. Of these, 128 + 32 = 160 were accessed for each of the first, second and third samples, while 127 + 32 = 159 were accessed for the fourth. In this way, the output of the Log VFD turns out to be exactly what was expected and confirms the correctness of the extension.

Record	Number of calls
0-8 (8 bytes) (H5FD_MEM_SUPER) Allocated	639
0-7 (8 bytes) (H5FD_MEM_SUPER) Read	639
8-16 (8 bytes) (H5FD_MEM_SUPER) Allocated	639
0-15 (16 bytes) (H5FD_MEM_SUPER) Read	639
16-96 (80 bytes) (H5FD_MEM_SUPER) Allocated	639
16-95 (80 bytes) (H5FD_MEM_SUPER) Read	639
96- 268437536 (268437440 bytes) (H5FD_MEM_DEFAULT) Allocated	639
96-607 (512 bytes) (H5FD_MEM_OHDR) Read	639
680-1191 (512 bytes) (H5FD_MEM_LHEAP) Read	639
136-679 (544 bytes) (H5FD_MEM_BTREE) Read	639
1072-1399 (328 bytes) (H5FD_MEM_BTREE) Read	639
800-1311 (512 bytes) (H5FD_MEM_OHDR) Read	639
2048-67110911 (67108864 bytes) (H5FD_MEM_DRAW) Read	160
67110912- 134219775 (67108864 bytes) (H5FD_MEM_DRAW) Read	160
134219776- 201328639 (67108864 bytes) (H5FD_MEM_DRAW) Read	160
201328640- 268437503 (67108864 bytes) (H5FD_MEM_DRAW) Read	159

Table 6.1.: Records and their number in each received log file for each extension configuration and DLIO Benchmark.

6.3.2. Performance comparison

As previously outlined in Section 4.3, in addition to a comparison of Log VFD logs for implementation validation, an examination was conducted of the execution time of the extension and DLIO Benchmark on the Lustre and BeeOND parallel file systems. The results of this comparison are presented in Figure 6.6. Furthermore, in addition to the runtime, the network bandwidth reported by the cluster's performance monitoring was also examined in order to ascertain its utilisation. The Lustre file system in CLAIX-23 is a shared resource by default; therefore, the experiment was conducted on a node on which other jobs could be run in parallel. It is therefore anticipated that the runtime and network bandwidth volatility in Lustre will be more significant than in BeeOND, where the experiment was conducted in isolated nodes. Furthermore, the data indicates that the Lustre and DLIO Benchmark exhibit a higher fabric bandwidth, irrespective of the number of read threads, whereas the BeeOND results are more definitive. The data also indicates that the extension is two to four times faster than the original DLIO Benchmark with varying numbers of read threads. Furthermore, it was observed that the runtime of the DLIO Benchmark increased with the utilisation of read threads, whereas the peak fabric bandwidth exhibited a decline. This phenomenon may be attributed to the overhead incurred by the creation of workers in high-level machine learning libraries.

Furthermore, the Unet3D workload simulation in Figure 6.7 provides additional assurance of the implementation, as the runtime difference remains constant in the BeeOND file system. This is to be expected, as the objective is not to achieve a perfect one-to-one replication of the DLIO workload, which is a repetition of existing work and is contrary to the goal of optimising the benchmarking of AI workloads. The data also demonstrates that the runtime of the extension is half as long, indicating that the implementation is twofold more efficient than the DLIO

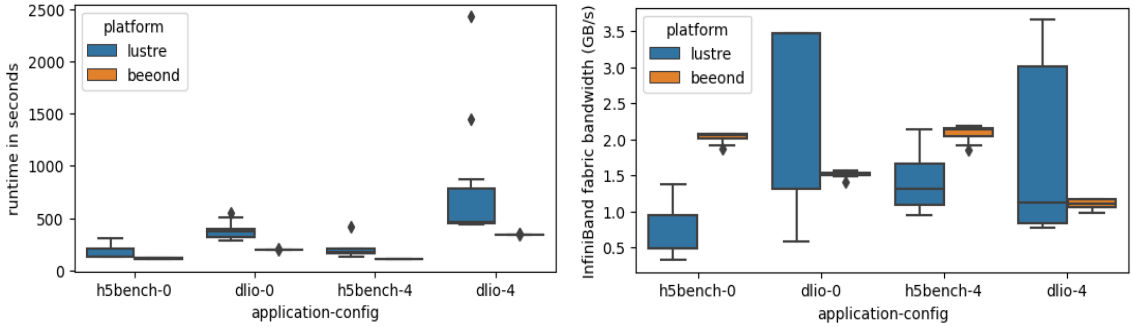

Figure 6.6.: Performance of the h5bench extension and DLIO Benchmark for the Lustre and BeeOND file systems. MPI launcher is `mpiexec`.

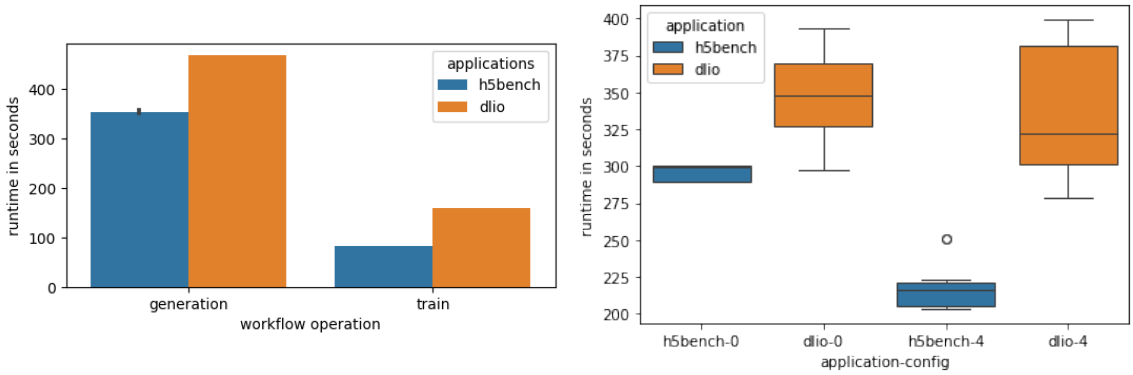

Figure 6.7.: Comparison of h5bench extension and DLIO Benchmark at emulating Unet3D workload (left) and performance when using read threads (right).

Benchmark. It is also noteworthy that in both the extension and DLIO Benchmark cases, the majority of the workflow time is allocated to data generation, which is an unsurprising outcome given that write operations frequently require more time than read operations.

Given the potential for MPI launchers to significantly impact runtime performance in CLAIX-23, an additional test was conducted utilising the `srun` launcher, as illustrated in Figure 6.7. From this result, it can be seen that the number of read threads has a more pronounced impact on execution time compared to the runtime observed in Figure 6.6. Moreover, this result corroborates the assertion that the DLIO Benchmark exhibits suboptimal scaling when the number of read threads is low, potentially leading to a decline in performance. These results also serve to confirm that the configurations in the h5bench extension are functioning as intended.

6.4. Extension performance evaluation

This section presents the results of scaling tests for the extension performed on a single node using 2^n MPI ranks. These results help determine if the extension performs as expected and showcase the CLAIX-23 HPC system's I/O performance for AI workloads using HDF5. Sections 6.4.1 and 6.4.2 will examine the performance of two MPI-IO modes, namely independent and collective. Sections 6.4.3 and 6.4.4 will present performance results employing HDF5-specific features, namely chunking, compression, and Subfiling VFD.

6.4.1. Independent mode

MPI independent mode is an approach to parallel computing in which different processes run independently and operate on separate data sets. In independent MPI mode, each process operates autonomously but can still interact with others, offering more flexibility.

The results of the scaling test in Figure 6.8 demonstrate a decline in performance per MPI rank, likely due to overhead from parallel access to the same HDF5 files. Given that the sequence of samples is randomly selected, it is plausible that multiple MPI ranks may require concurrent access to the same file. It is also important to note that the test is measuring strong scaling, which occurs when the number of MPI ranks increases but the amount of data remains constant. This may result in a greater number of parallel accesses. When the total performance is taken into account, it can be concluded that the read rate stops increasing significantly after 16 MPI ranks.

The graph shows also peak fabric bandwidth for different MPI rank counts. Despite high read rates, peak bandwidth declines, which may seem counterintuitive. This is due to the fact that the fabric bandwidth was obtained through the utilisation of a job monitoring tool, namely Perfmon [84], which is available on the CLAIX-23 cluster and is configured such that the time between each fabric bandwidth measurement is 1 minute. However, when 32 and 64 MPI ranks were involved, the task duration was less than a minute. Therefore, it can be postulated that the peak bandwidth may not be attributed to the training simulation itself, but rather to the generation of data.

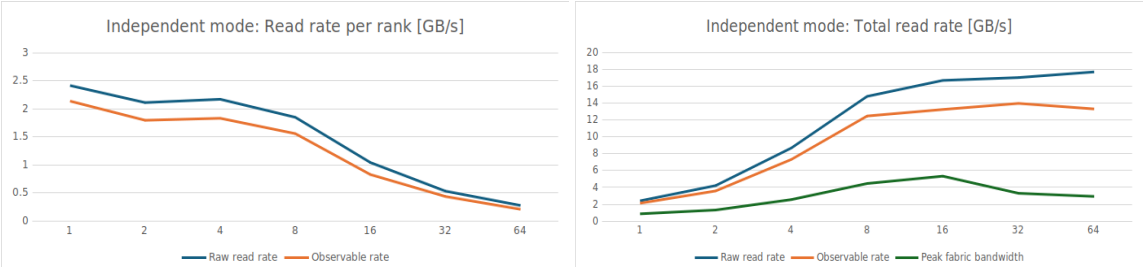

Figure 6.8.: Scaling test results for MPI independent mode on BeeOND.

6.4.2. Collective mode

In the context of MPI, the term "collective mode" is used to describe a method of operation in which multiple processes collaborate in a coordinated manner. In contrast to the independent mode, in which each process operates independently, the collective mode necessitates the collaboration of all participating processes in order to achieve a single task.

Nevertheless, as illustrated in Figure 6.9, the utilisation of collective mode has been observed to result in a performance decline of up to 1 GB/s in comparison to independent mode, when considering the read rate per MPI rank. This can be explained by the fact that, in addition to the overhead of parallel file access, there is also the overhead of communication between all MPI ranks. This results in a total performance degradation of up to 5 GB/s compared to an independent mode for the same number of MPI ranks. It is noteworthy that the maximum performance gain is also observed when using up to 16 MPI ranks, after which it ceases to increase significantly. In this experiment, Perfmon was also employed to ascertain the peak fabric bandwidth. As a consequence of the limitations described in the previous section, a decline in performance was also observed for 32 and 64 MPI ranks.

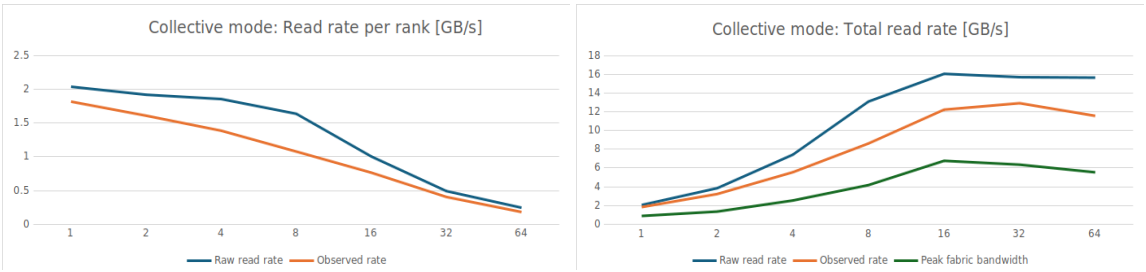

Figure 6.9.: Scaling test results for MPI collective mode on BeeOND.

6.4.3. Chunking and compressing

The technique of chunking in HDF5 involves the division of a large dataset into smaller, fixed-size blocks, or "chunks", which are then stored separately within a file. This results in more efficient data access, particularly when working with large multidimensional datasets where only a portion of the data is required at any given time. The retrieval of only the necessary chunks, as opposed to the entire dataset, can result in a notable enhancement in I/O performance, particularly when data compression is employed.

As illustrated in Figure 6.10, the read rate per rank remains relatively constant regardless of the number of MPI ranks involved. This indicates a satisfactory level of scalability, which can be attributed to the absence of overheads associated with communication between MPI ranks and the absence of overheads for parallel access to HDF5 files by multiple MPI ranks. Furthermore, the total read rate demonstrates excellent scalability, exhibiting an exponential growth pattern. At the peak, the observed rate reached 25 GB/s, while the raw read rate reached 45 GB/s for 64

MPI ranks. This is a significant improvement over the independent MPI mode, which exhibited a peak rate of 14 GB/s and 18 GB/s, respectively.

As compression in HDF5 is contingent upon chunking, I proceeded to enable compression in the aforementioned configuration by setting the compression level to 4. The results, as illustrated in Figure 6.11, also demonstrate good scalability, which is a key benefit of chunking. However, given that the data for the experiment was generated randomly, it was possible to achieve a relatively poor compression rate (less than 1%), which resulted in almost identical outcomes.

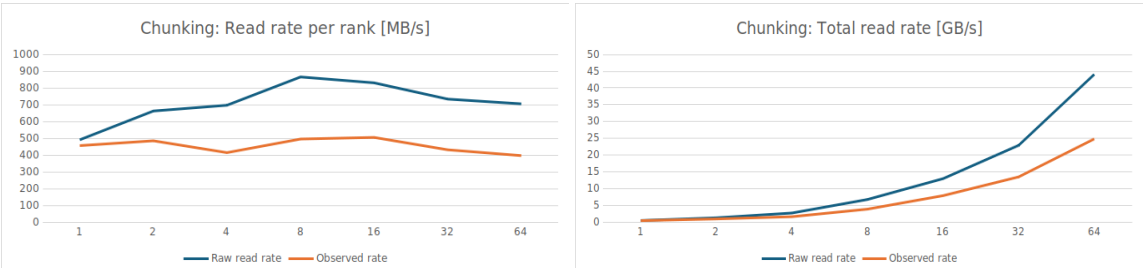

Figure 6.10.: Scaling test results of chunking on Lustre.

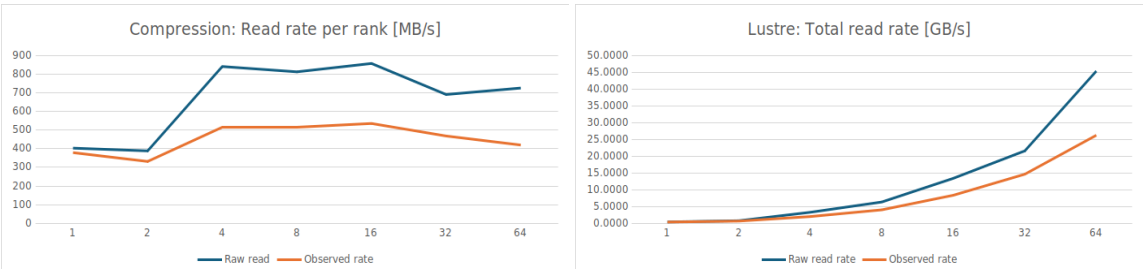

Figure 6.11.: Scaling test results of level 4 compression on Lustre.

6.4.4. Subfiling VFD

Subfiling VFD represents a further feature of HDF5, designed to enhance the performance of data storage and retrieval on parallel file systems. Subfiling entails the division of a single, large HDF5 file into multiple, smaller subfiles. This distribution has the effect of reducing I/O contention and enhancing data access speeds, as it allows multiple processes to read from or write to different subfiles simultaneously. It is anticipated that Subfiling VFD will optimise the scalability and performance of applications utilising HDF5.

Nevertheless, as illustrated in Figure 6.12, during the experiment, Subfiling VFD exhibited suboptimal scalability with a notable decline in performance when utilising a substantial number of MPI ranks. When employing 64 MPI ranks, the observed rate was a mere 800 MB/s, which is considerably below the expected threshold. It seems reasonable to posit that the observed suboptimal performance may be attributable to the fact that the default Subfiling VFD settings were employed, and

that more fine-tuning may be required. For instance, the issue may be attributable to the fact that, despite the settings indicating that each subfile should contain a single sample, in practice, some files may contain fragments of multiple samples. This could necessitate the opening and reading of multiple subfiles, thereby increasing the overhead associated with parallel file access. This phenomenon requires further investigation.

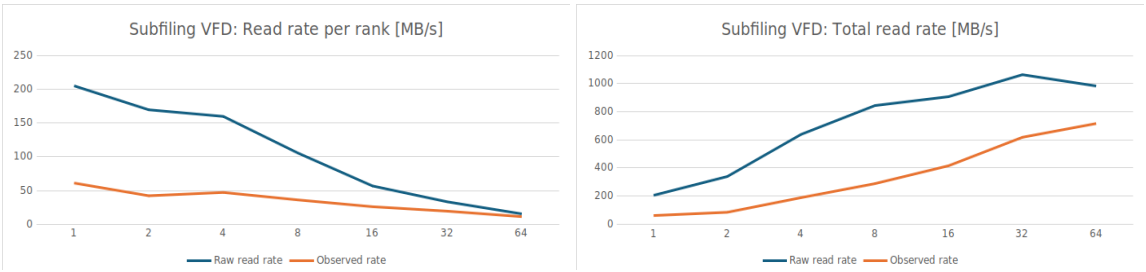

Figure 6.12.: Scaling test results of Subfiling VFD on Lustre.

7. Conclusion

As a result of this work, an extension to h5bench has been developed that successfully mimics the AI workload of DLIO Benchmark. This is confirmed by the fact that the Log VFD results being identical for different DLIO Benchmark and extension configurations, leading to the conclusion that the I/O patterns are identical from the perspective of the HDF5 file. Studying the behaviour of the DLIO Benchmark and the machine learning libraries PyTorch and TensorFlow with the help of the DFTracer profiler allowed to get a more complete and detailed picture of the processes occurring when working with the files and I/O patterns generated in this case. This revealed the peculiarities of the read thread configuration when using PyTorch and Tensorflow, consisting in the number of threads actually created for read threads being equal to zero.

Also, runtime measurements for the DLIO Benchmark and the extension using different numbers of read threads and for different parallel file systems, namely Lustre and BeeOND, showed that the runtime of the extension was 1.5-4 times less in all cases. Perfmon peak fabric bandwidth measured for the nodes on which the experiment was run showed that the extension also performed better. This observation is attributed to the discrepancy in the overhead introduced by the C application in the extension and the Python application in the original benchmark. While a quicker runtime may be beneficial for faster testing, it also suggests that the benchmark may not fully capture the complexity of real AI workloads, such as high metadata operations introduced by the use of Python-based libraries.

The results of the scaling tests demonstrate that the benchmark functions as expected, with the exception of the case involving Subfiling VFD, which requires further investigation. Of particular note are the scaling tests using chunking, an HDF5 specific feature, which also showed an expectedly high result achieved by optimising partial I/O.

The objective of ensuring the benchmark suite and configurations are capable of functioning correctly to test both AI and traditional HPC workloads also has been met. With the inclusion of AI workload into h5bench, users can streamline the HDF5 benchmarking process for their HPC systems.

8. Future work

The prototype represents merely the initial stage of the project, as numerous additional features have the potential to enhance the capabilities of AI applications.

It is acknowledged that the primary objective of this benchmark extension is to assess the throughput of the read I/O operation, with only a limited scope of coverage dedicated to metadata time in Section 5.3. It is possible to extend the coverage of the performance of the metadata in the future. Additionally, the benchmark does not provide any information regarding the system's write performance, which is a consequence of the nature of the training process and model evaluation. It should be noted, however, that there is another process involved in AI workloads, namely checkpointing. This function is responsible for storing the model weights in non-volatile memory during the training process. Consequently, the benchmark could also provide data regarding the write performance of a checkpointing simulation.

An additional method for enhancing the capabilities of the benchmark is to incorporate additional parameters, thereby enabling the emulated machine learning workflow to be further refined. To illustrate, when training machine learning models on files containing images, it is not always the case that the entire image is the same. Therefore, in the real world, the sample size may vary. To address these potential issues, it would be possible to introduce a parameter defining the mean and standard deviation for each generated sample. Furthermore it would be feasible to analyse and add support for I/O pattern emulation of libraries and machine learning frameworks other than PyTorch and TensorFlow, such as Keras [51], Scikit-Learn [68] or NVIDIA Data Loading Library (DALI) [23].

As previously stated, h5bench currently offers support for a number of VOL Connectors and VFDs, including Async, Cache, Log VOLs and Subfiling VFD. While the latter has already been implemented, support for the former three is currently only planned for the future. The Async VOL was designed with the objective of supporting asynchronous I/O for the parallelisation of computations and work with the HDF5 API. This has the potential to result in significant performance improvements. The operation of a Log VOL is as follows: write requests from an MPI process are appended one after another (as logs) in a contiguous space within the file. It is anticipated that this approach will enhance the performance of write operations. [39] However, until the checkpointing functionality is incorporated into the benchmark, it is not yet feasible to extend support for this VOL. Another topic of interest is the Cache VOL, which provides enhanced cache availability. As the utilisation of Cache VOL does not necessitate any alterations to the code and is solely connected through environment variables [36], it is anticipated that the extension is already capable of supporting this VOL. However, this assertion requires

confirmation.

As previously stated, the extension already incorporates compression support, albeit currently constrained to utilising the zlib library. This compression method is a standard feature of HDF5; however, alternative I/O filters, including BZIP2 [35], LZO [49] and LZ4 [10], can be employed as supplementary options. A comprehensive index of all available filters can be accessed on the HDF5 Group support portal.

Nevertheless, in addition to enhancing the benchmark itself, there is also scope to investigate the distinctive characteristics of certain HPC systems and the diverse factors and parameters that influence the performance of AI workloads. For instance, this thesis did not examine the metadata rate data obtained for different configurations. Additionally, the experiments did not examine the impact of varying parameters, such as sample size, the number of samples per file, and the number of training files, on performance. The question of the observed decline in I/O performance when using Subfiling VFD also remains open. All these circumstances present a significant opportunity for research activities in the future.

This extension work is currently under review for pull request in: `https://github.com/arcturus5340/h5bench`.

A. h5bench extension parameters

Parameter	Description	Type	Default
generate-data	Enable generation of benchmarking data	bool	false
train	Enable model training simulation	bool	false
evaluation	Enable model evaluation simulation	bool	false
record-length	Record size of a single sample in bytes	int	67108864
num-files-train	The number of files used to train the model	int	32
num-files-eval	The number of files used to evaluate the model	int	8
num-samples-per-file	The number of samples in each file	int	4
file-prefix	Prefix in the name of files containing training and evaluation data	string	img
chunking	Enable chunking	bool	false
chunk-size	Chunk size	int	1024
keep-files	Does not delete data after the benchmark is finished	bool	false
compression	Enable compression	bool	false
compression-level	Compression level from 1 to 9	int	4
batch-size	Training batch size	int	7
batch-size-eval	Evaluation batch size	int	2
shuffle	Enable samples shuffle	bool	false
preprocess-time	Preprocessing time after reading each sample in seconds	float	0.0
preprocess-time-stdev	Standard deviation in preprocessing time in seconds	float	0.0
epochs	The number of epochs	int	5
total-training-steps	Maximum number of steps per training per epoch	int	-1
computation-time	Computation time after reading each batch in seconds	float	0.323
computation-time-stdev	Standard deviation in computation time in seconds	float	0.0
random-seed	Random seed to be used	int	42
eval-time	Evaluation time after reading each batch in seconds	float	0.323
eval-time-stdev	Standard deviation in evaluation time in seconds	float	0.0
epochs-between-evals	The number of epochs between evaluations	int	1
train-data-folder	Name of the directory containing the training data	string	train
valid-data-folder	Name of the directory containing the validation data	string	valid
records-dataset-name	Name of the dataset with records	string	records
labels-dataset-name	Name of the dataset with labels	string	labels
seed-change-epoch	Enable seed changes every epoch	bool	false
read-threads	The number of workers used to read the data	int	4
collective-meta	Enable collective HDF5 metadata operations	bool	false
collective-data	Enable collective HDF5 data operations	bool	false
subfiling	Enable HDF5 Subfiling Virtual File Driver	bool	false
output-csv-name	Name of the output csv file	string	output
output-ranks-data	Enable statistics output for each rank	bool	false

Table A.1.: The h5bench extension parameters.

B. Benchmark configurations

	Configuration		
	A	B	C
record-length	67108864	146600628	67108864
num-files-train	128	168	512
num-files-eval	32	0	128
num-samples-per-file	4	4	4
batch-size	7	4	7
batch-size-eval	2	0	2
read-threads	0/4	4	0/4
computation-time	0.323	1.3604	0.323
eval-time	0.323	0.0	0.323
collective-meta	true	true	true
collective-data	true	true	true
chunking	false	false	false
compression-level	-	-	-
subfiling	false	false	false

Table B.1.: Configurations for h5bench extension and DLIO Benchmark performance comparison tests.

	Configuration				
	D	E	F	G	H
record-length	67108864	67108864	67108864	67108864	67108864
num-files-train	256	256	256	256	256
num-files-eval	64	64	64	64	64
num-samples-per-file	4	4	4	4	4
batch-size	7	7	7	7	7
batch-size-eval	2	2	2	2	2
read-threads	0	0	0	0	0
computation-time	0.323	0.323	0.323	0.323	0.323
eval-time	0.323	0.323	0.323	0.323	0.323
collective-meta	false	true	false	false	false
collective-data	false	true	false	false	false
chunking	false	false	true	true	false
compression-level	-	-	-	4	-
subfiling	false	false	false	false	true

Table B.2.: Configurations for the h5bench extension used in the scaling tests.

C. Experimental data

Note: FBW stands for Fabric Bandwidth and SD for Standard Deviation.

	0 read threads				4 read threads			
	h5bench		DLIO Benchmark		h5bench		DLIO Benchmark	
	Runtime	FBW	Runtime	FBW	Runtime	FBW	Runtime	FBW
1. measurement	02:37.5	1.11	05:59.9	1.39	03:22.6	1.31	07:23.9	0.78
2. measurement	02:10.5	0.48	06:03.0	0.69	02:37.6	2.14	07:37.8	0.86
3. measurement	02:10.4	0.48	09:15.3	0.58	02:18.2	1.33	07:30.8	1.12
4. measurement	02:10.3	0.48	04:46.7	3.48	03:23.6	1.01	07:27.7	0.80
5. measurement	02:10.1	0.48	06:39.0	3.48	02:58.3	0.96	14:35.1	3.23
6. measurement	03:47.5	0.48	06:29.9	3.48	02:58.8	1.72	07:38.1	1.13
7. measurement	04:40.8	0.48	06:29.7	3.48	06:59.0	1.08	24:05.3	3.66
8. measurement	05:10.9	0.32	05:07.7	3.48	03:26.4	1.14	07:21.6	0.83
9. measurement	02:11.1	1.34	05:07.9	3.48	02:58.1	1.74	40:31.3	3.09
10. measurement	02:10.6	1.38	08:28.4	1.29	02:13.8	1.46	08:38.1	2.78
Minimum	02:10.1	0.32	04:46.7	0.58	02:13.8	0.96	07:21.6	0.78
Maximum	05:10.9	1.38	09:15.3	3.48	06:59.0	2.14	40:31.3	3.66
Average	02:56.0	0.70	06:26.7	2.48	03:19.6	1.39	13:17.0	1.83
SD	01:10.4	0.40	01:26.4	1.31	01:21.2	0.38	10:58.1	1.20

Table C.1.: Performance comparison of h5bench extension and DLIO Benchmark for different number of read-threads using `mpiexec` MPI launcher on Lustre.

	0 read threads				4 read threads			
	h5bench		DLIO Benchmark		h5bench		DLIO Benchmark	
	Runtime	FBW	Runtime	FBW	Runtime	FBW	Runtime	FBW
1. measurement	01:53.8	2.06	03:14.1	1.51	01:47.0	2.14	05:40.1	1.16
2. measurement	01:54.2	2.05	03:14.0	1.53	01:48.2	2.15	05:42.0	0.98
3. measurement	01:51.9	2.07	03:14.9	1.51	01:46.6	2.17	05:41.8	1.06
4. measurement	01:55.1	1.92	03:20.3	1.55	01:45.7	2.19	05:41.9	1.06
5. measurement	01:56.6	1.87	03:25.9	1.41	01:44.5	2.07	05:45.8	1.06
6. measurement	01:56.2	1.99	03:16.2	1.48	01:47.8	2.04	05:41.5	1.17
7. measurement	01:51.5	2.08	03:15.7	1.53	01:45.8	1.91	05:42.0	1.17
8. measurement	01:52.1	2.07	03:15.2	1.56	01:48.2	1.85	05:42.9	1.16
9. measurement	01:52.5	2.05	03:15.2	1.54	01:45.7	2.14	05:42.4	1.17
10. measurement	01:52.8	2.07	03:16.8	1.52	01:46.4	2.14	05:45.4	1.04
Minimum	01:51.5	1.87	03:14.0	1.41	01:44.5	1.85	05:40.1	0.98
Maximum	01:56.6	2.08	03:25.9	1.56	01:48.2	2.19	05:45.8	1.17
Average	01:53.7	2.02	03:16.8	1.51	01:46.6	2.08	05:42.6	1.10
SD	00:01.8	0.07	00:03.7	0.04	00:01.2	0.12	00:01.8	0.07

Table C.2.: Performance comparison of h5bench extension and DLIO Benchmark for different number of read-threads using `mpiexec` MPI launcher on BeeOND.

	h5bench		DLIO Benchmark	
	Gen runtime	Train runtime	Gen runtime	Train runtime
1. measurement	05:53.4	01:22.0	07:46.8	02:38.8
2. measurement	05:51.1	01:22.1	07:49.1	02:39.4
3. measurement	05:53.0	01:22.6	07:47.4	02:39.2
4. measurement	06:08.7	01:22.2	07:48.3	02:39.3
5. measurement	05:53.0	01:22.9	07:50.1	02:39.8
6. measurement	05:48.6	01:23.7	07:48.7	02:40.0
7. measurement	05:51.3	01:21.8	07:50.0	02:39.0
8. measurement	05:54.6	01:21.2	07:47.1	02:40.0
9. measurement	05:50.6	01:22.2	07:47.2	02:38.8
10. measurement	05:57.1	01:22.4	07:46.2	02:40.7
Minimum	0:05:49	0:01:21	0:07:46	0:02:39
Maximum	0:06:09	0:01:24	0:07:50	0:02:41
Average	0:05:54	0:01:22	0:07:48	0:02:40
SD	0:00:06	0:00:01	0:00:01	0:00:01

Table C.3.: Performance comparison of h5bench extension and DLIO Benchmark in Unet3D workload simulation using `mpiexec` MPI launcher on Lustre.

	0 read threads		4 read threads	
	h5bench	DLIO Benchmark	h5bench	DLIO Benchmark
1. measurement	06:33.2	03:42.8	03:42.8	05:38.0
2. measurement	05:27.3	03:25.8	03:25.8	05:01.6
3. measurement	05:27.4	04:10.9	04:10.9	06:39.1
4. measurement	05:27.4	04:10.9	04:10.9	06:29.4
5. measurement	05:27.4	03:25.7	03:25.7	06:29.4
6. measurement	06:09.0	03:23.4	03:23.4	05:02.8
7. measurement	04:57.7	03:36.6	03:36.6	04:38.4
8. measurement	06:09.5	03:36.5	03:36.5	04:38.4
9. measurement	06:09.5	03:36.5	03:36.5	05:56.2
10. measurement	06:09.1	03:23.5	03:23.5	05:41.3
Minimum	04:49.3	04:57.7	03:23.4	04:38.4
Maximum	05:00.7	06:33.2	04:10.9	06:39.1
Average	04:55.6	05:47.8	03:39.3	05:37.5
SD	00:05.4	00:30.0	00:18.0	00:46.0

Table C.4.: Performance comparison of h5bench extension and DLIO Benchmark for different number of read-threads using `srun` MPI launcher on Lustre.

Ranks	Avg. raw read rate per rank	Avg. observed read rate per rank	Total raw read rate	Total observed read rate	FBW
1	2.4148	2.1374	2.4148	2.1374	0.8753
2	2.1110	1.7958	4.2220	3.5916	1.3280
4	2.1708	1.8308	8.6832	7.3232	2.5660
8	1.8492	1.5580	14.7936	12.4640	4.4700
16	1.0428	0.8275	16.6848	13.2400	5.3420
32	0.5319	0.4363	17.0214	13.9622	3.3120
64	0.2765	0.2078	17.6960	13.3018	2.9400

Table C.5.: Results of scaling tests using MPI independent mode on BeeOND.

Ranks	Avg. raw read rate per rank	Avg. observed read rate per rank	Total raw read rate	Total observed read rate	Max FBW
1	2.0370	2.0370	1.8160	1.8160	0.7519
2	1.9182	3.8364	1.6080	3.2160	1.3220
4	1.8540	7.4160	1.3850	5.5400	2.4500
8	1.6360	13.0880	1.0765	8.6118	4.0220
16	1.0030	16.0480	0.7637	12.2195	5.1320
32	0.4904	15.6921	0.4034	12.9088	5.6620
64	0.2444	15.6429	0.1806	11.5571	4.2200

Table C.6.: Results of scaling tests using MPI collective mode on BeeOND.

Ranks	Avg. raw read rate per rank	Avg. observed read rate per rank	Total raw read rate	Total observed read rate
1	205.0292	205.0292	60.9944	60.9944
2	169.5078	339.0156	42.1616	84.3232
4	159.6886	638.7544	47.0522	188.2088
8	105.4838	843.8704	36.0164	288.1312
16	56.7026	907.2416	25.9826	415.7216
32	33.2454	1063.8528	19.3200	618.2400
64	15.3638	983.2832	11.1892	716.1088

Table C.7.: Results of scaling tests using Subfiling VFD on Lustre.

Ranks	Avg. raw read rate per rank	Avg. observed read rate per rank	Total raw read rate	Total observed read rate
1	491.2370	0.4797	456.5632	0.4459
2	662.6882	1.2943	485.2470	0.9477
4	696.9758	2.7225	414.8060	1.6203
8	865.4988	6.7617	495.7740	3.8732
16	830.3256	12.9738	505.2046	7.8938
32	733.5030	22.9219	431.6474	13.4890
64	705.2514	44.0782	397.0922	24.8183

Table C.8.: Results of scaling tests using chunking on Lustre.

Ranks	Avg. raw read rate per rank	Avg. observed read rate per rank	Total raw read rate	Total observed read rate
1	402.5700	0.3931	378.6344	0.3698
2	387.9576	0.7577	331.2904	0.6471
4	840.2202	3.2821	513.0336	2.0040
8	811.5844	6.3405	513.1686	4.0091
16	856.5122	13.3830	534.8222	8.3566
32	690.1876	21.5684	467.7842	14.6183
64	724.8768	45.3048	419.4482	26.2155

Table C.9.: Results of scaling tests using chunking and compression level 4 on Lustre.

Bibliography

[1] Introduction - ior 4.1.0+dev documentation. `https://ior.readthedocs.io/en/latest/index.html`. [Accessed 2024-08-08].

[2] MLCommons. `https://mlcommons.org/`. [Accessed 2024-08-09].

[3] Pytorch documentation. `https://pytorch.org/docs/stable/data.html#multi-process-data-loading`. [Accessed 2024-09-06.

[4] R. Adolf, S. Rama, B. Reagen, G.-Y. Wei, and D. Brooks. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[5] L. Aithani, E. Alcaide, S. Bartunov, C. D. Cooper, A. S. Doré, T. J. Lane, F. Maclean, P. Rucktooa, R. A. Shaw, and S. E. Skerratt. Advancing structural biology through breakthroughs in ai. *Current Opinion in Structural Biology*, 80:102601, 2023.

[6] A. L. C. F. ALCF. Deep learning i/o (dlio) benchmark on github. `https://github.com/argonne-lcf/dlio_benchmark/tree/main/dlio_benchmark/configs/workload`. [Accessed 2024-09-11].

[7] Apache Mesos. RecordIO: Apache Mesos Documetation.

[8] Apache Software Foundation. Apache Parquet.

[9] O. Balmau. Characterizing i/o in machine learning with mlperf storage. 51(3), 2022.

[10] M. Bartík, S. Ubik, and P. Kubalik. Lz4 compression algorithm on fpga. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 179–182. IEEE, 2015.

[11] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler. A modular benchmarking infrastructure for high-performance and reproducible deep learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 66–77. IEEE, 2019.

[12] J. L. Bez, S. Byna, and S. Ibrahim. I/o access patterns in hpc applications: A 360-degree survey. *ACM Computing Surveys*, 56(2):1–41, 2023.

[13] J. L. Bez, H. Tang, S. Breitenfeld, H. Zheng, W. Liao, K. Hou, Z. Huang, and S. Byna. h5bench: A unified benchmark suite for evaluating hdf5 i/o performance on pre-exascale platforms. *Concurrency and Computation. Practice and Experience*, 36(16), 4 2024.

[14] S. Byna. Efficient parallel i/o with hdf5 and proactive data containers (pdc). `https://www.olcf.ornl.gov/wp-content/uploads/2019/08/20190625_Suren_OLCF-talk-s.pdf`, 2019. [Accessed 2024-06-29].

[15] S. Byna, M. S. Breitenfeld, B. Dong, Q. Koziol, E. Pourmal, D. Robinson, J. Soumagne, H. Tang, V. Vishwanath, and R. Warren. Exahdf5: Delivering efficient parallel i/o on exascale computing systems. *Journal of Computer Science and Technology*, 35(1), 1 2020.

[16] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin, 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*, pages 1–12. IEEE, 2012.

[17] S. Byna, M. Howison, and A. Sim. Parallel i/o kernel (piok) suite. *Accessed: Mar*, 16:2020, 2015.

[18] P. Carns. Darshan. In *High performance parallel I/O*, pages 351–358. Chapman and Hall/CRC, 2014.

[19] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[20] F. Chowdhury, Y. Zhu, T. Heer, S. Paredes, A. Moody, R. Goldstone, K. Mohror, and W. Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[21] P. Colella, D. Graves, J. Johnson, H. Johansen, N. Keen, T. Ligocki, D. Martin, P. McCorquodale, D. Modiano, P. Schwartz, et al. Chombo software package for amr applications design document. *Lawrence Berkeley National Laboratory, Berkeley, CA*, 2012.

[22] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. Dawnbench: An end-to-end deep learning benchmark and competition. *Training*, 100(101):102, 2017.

[23] N. Corporation. Nvidia data loading library (dali). `https://developer.nvidia.com/dali`. [Accessed 2024-09-10].

[24] B. Dally. Hardware for deep learning. In *2023 IEEE Hot Chips 35 Symposium (HCS)*, pages 1–58. IEEE Computer Society, 2023.

[25] M. R. N. Darbandi, M. Darbandi, S. Darbandi, I. Bado, M. Hadizadeh, and H. R. K. Khorshid. Artificial intelligence breakthroughs in pioneering early diagnosis and precision treatment of breast cancer: A multimethod study. *European Journal of Cancer*, page 114227, 2024.

[26] H. Devarajan. Dftracer. `https://github.com/hariharan-devarajan/dftracer`, 2024. [Accessed 2024-06-27].

[27] H. Devarajan, A. Kougkas, H. Zheng, V. Vishwanath, and X.-H. Sun. Stimulus: Accelerate data management for scientific ai applications in hpc. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 109–118. IEEE, 2022.

[28] H. Devarajan, H. Zheng, A. Kougkas, X.-H. Sun, and V. Vishwanath. Dlio: A data-centric benchmark for scientific deep learning applications. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 81–91, 2021.

[29] D. Djebarov. Issue on the darshan project's github page describing startup problems. `https://github.com/darshan-hpc/darshan/issues/989`, 2024. [Accessed 2024-09-06].

[30] J. L. B. et al. Unbalanced workload plot. `https://dxt-explorer.readthedocs.io/en/latest/unbalanced-ranks.html`, 2022. [Accessed 2024-09-10].

[31] R. L. et al. The parallel netcdf, 2010.

[32] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 workshop on array databases*, pages 36–47, 2011.

[33] J.-l. Gailly and M. Adler. Zlib compression library. 2004.

[34] W. Gao, F. Tang, L. Wang, J. Zhan, C. Lan, C. Luo, Y. Huang, C. Zheng, J. Dai, Z. Cao, et al. Aibench: an industry standard internet service ai benchmark suite. *arXiv preprint arXiv:1908.08998*, 2019.

[35] J. Gilchrist. Parallel data compression with bzip2. In *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, volume 16, pages 559–564. Citeseer, 2004.

[36] T. H. Group. Hdf5 cache vol: Efficient parallel i/o through caching data on fast storage layers. `https://github.com/HDFGroup/vol-cache`. [Accessed 2024-09-10].

[37] T. H. Group. Hdf5 user's guide. https://portal.hdfgroup.org/documentation/hdf5-docs/advanced_topics/chunking_in_hdf5.html. [Accessed 2024-09-10].

[38] T. H. Group. Log vfd. https://docs.hdfgroup.org/hdf5/v1_14/group___f_a_p_l.html#ga4e03be2fe83ed02b32266a6c81427beb. [Accessed 2024-09-06.

[39] T. H. Group. Log vol - an hdf5 vol connector for storing data in a time-log layout in files. https://github.com/HDFGroup/vol-log-based. [Accessed 2024-09-10].

[40] T. H. Group. Hdf software documentation. https://davis.lbl.gov/Manuals/HDF5-1.8.7/UG/UG_frame10Datasets.html, 2011. [Accessed 2024-01-08].

[41] T. H. Group. Hdf5 file format specification version 3.0. https://docs.hdfgroup.org/hdf5/v1_14/_f_m_t3.html, 2022. [Accessed 2024-07-11].

[42] T. H. Group. Hdf5: Api reference v1.14.3. https://docs.hdfgroup.org/hdf5/v1_14, 2023. [Accessed 2024-01-08].

[43] A. Hannun. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567*, 2014.

[44] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[45] J. Henderson. Hdf5 subfiling vfd userâs guide. Technical report, The HDF5 Group, 2023.

[46] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[47] R. W. Houim and E. S. Oran. A technique for computing dense granular compressible flows with shock waves. *arXiv preprint arXiv:1312.1290*, 2013.

[48] S. James, Z. Ma, D. R. Arrojo, and A. J. Davison. Rlbench: The robot learning benchmark & learning environment. *IEEE Robotics and Automation Letters*, 5(2):3019–3026, 2020.

[49] J. Kane and Q. Yang. Compression speed enhancements to lzo for multi-core systems. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 108–115. IEEE, 2012.

[50] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

[51] N. Ketkar and N. Ketkar. Introduction to keras. *Deep learning with python: a hands-on introduction*, pages 97–111, 2017.

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

[53] J. Kunkel, J. Bent, J. Lofstead, and G. S. Markomanolis. Establishing the io-500 benchmark. *White Paper*, 2016.

[54] N. Lewis, J. L. Bez, and S. Byna. I/o in machine learning applications on hpc systems: A 360-degree survey. *arXiv preprint arXiv:2404.10386*, 2024.

[55] T. Li, S. Byna, Q. Koziol, H. Tang, J. L. Bez, and Q. Kang. h5bench: Hdf5 i/o kernel suite for exercising hpc i/o patterns. In *Proceedings of Cray User Group Meeting, CUG*, volume 2021, 2021.

[56] G. LLC. Perfetto. `https://ui.perfetto.dev/`, 2024. [Accessed 2024-06-27].

[57] W. Loewe, R. Hedges, T. McLarty, and C. Morrone. Llnl's parallel i/o testing tools and techniques for asc parallel file systems. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2004.

[58] S. Makridakis. The forthcoming artificial intelligence (ai) revolution: Its impact on society and firms. *Futures*, 90:46–60, 2017.

[59] Z. Masih. On demand file systems with beegfs. 2023.

[60] H. Mikami, H. Suganuma, Y. Tanaka, Y. Kageyama, et al. Massively distributed sgd: Imagenet/resnet-50 training in a flash. *arXiv preprint arXiv:1811.05233*, 2018.

[61] M. Miller. Multi-purpose, application-centric, scalable i/o proxy application. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), 2015.

[62] V. Mnih. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[63] N. Nahar, S. Zhou, G. Lewis, and C. Kästner. Collaboration challenges in building ml-enabled systems: communication, documentation, engineering, and process. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 413â425, New York, NY, USA, 2022. Association for Computing Machinery.

[64] NumPy. numpy.lib.format. `https://numpy.org/devdocs/reference/generated/numpy.lib.format.html`. [Accessed 2024-08-08].

[65] E. Párraga, B. León, R. Bond, D. Encinas, A. Bezerra, S. Mendez, D. Rexachs, and E. Luque. Analyzing the i/o patterns of deep learning applications. In *Cloud Computing, Big Data & Emerging Topics: 9th Conference, JCC-BD&ET, La Plata, Argentina, June 22-25, 2021, Proceedings 9*, pages 3–16. Springer, 2021.

[66] M. M. A. Patwary, S. Byna, N. R. Satish, N. Sundaram, Z. Lukić, V. Roytershteyn, M. J. Anderson, Y. Yao, Prabhat, and P. Dubey. Bd-cats: big data clustering at trillion particle scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[67] A. K. Paul, O. Faaland, A. Moody, E. Gonsiorowski, K. Mohror, and A. R. Butt. Understanding hpc application i/o behavior using system level statistics. In *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 202–211, 2020.

[68] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of machine Learning research*, 12:2825–2830, 2011.

[69] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020.

[70] P. Schwan et al. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux symposium*, volume 2003, pages 380–386, 2003.

[71] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[72] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, and N. J. Wright. Modular hpc i/o characterization with darshan. In *2016 5th workshop on extreme-scale programming tools (ESPT)*, pages 9–17. IEEE, 2016.

[73] S.Snyder. Example of darshan's pdf report. `https://www.S/research/projects/darshan/docs/ssnyder_ior-hdf5_id3655016_9-23-29011-12333993518351519212_1.darshan.pdf/`, 2015. [Accessed 2024-09-10].

[74] F. Su, C. Liu, and H.-G. Stratigopoulos. Testability and dependability of ai hardware: Survey, trends, challenges, and perspectives. *IEEE Design & Test*, 40(2):8–58, 2023.

[75] I. Sutskever. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

[76] H. Tang, Q. Koziol, J. Ravi, and S. Byna. Async vol connector repository. `https://github.com/HDFGroup/vol-async`. [Accessed 2024-09-06.

[77] H. Tang, Q. Koziol, J. Ravi, and S. Byna. Transparent asynchronous parallel i/o using background threads. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):891–902, 2021.

[78] TensorFlow. Tfrecord and tf.train.example. `https://www.tensorflow.org/tutorials/load_data/tfrecord`. [Accessed 2024-08-08].

[79] R. Thakur, W. Gropp, and E. Lusk. On implementing mpi-io portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32, 1999.

[80] The HDF Group. Hierarchical Data Format, version 5.

[81] J. Thiyagalingam, M. Shankar, G. Fox, and T. Hey. Scientific machine learning benchmarks. *Nature Reviews Physics*, 4(6):413–420, 2022.

[82] Top500. CLAIX-2023 (CPU segment) - NEC HPC1808Rk-2 DLC, Xeon Platinum 8468 48C 2.1GHz, Infiniband NDR | TOP500 — top500.org. `https://www.top500.org/system/180286/`. [Accessed 2024-08-08].

[83] L. UChicago Argonne. Dlio benchmark documentation. `https://dlio-benchmark.readthedocs.io/en/latest/config.html`. [Accessed 2024-09-11].

[84] I.-C. R. A. University. Perfmon job monitoring system. `https://help.itc.rwth-aachen.de/service/rhr4fjjutttf/article/3a11a76fdf30476bb4b1a8b30661dab3/`. [Accessed 2024-09-07].

[85] R. A. University. Claix-2023: New supercomputer at the rwth. `https://blog.rwth-aachen.de/itc/en/2024/02/09/claix-2023/`, 2024. [Accessed 2024-08-26].

[86] J. Weston, S. Chopra, and A. Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

[87] K. Wu, S. Byna, and B. Dong. Vpic io utilities. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2018.

[88] C. Xu, S. Snyder, V. Venkatesan, P. Carns, O. Kulkarni, S. Byna, R. Sisneros, and K. Chadalavada. Dxt: Darshan extended tracing. Technical report, Argonne National Lab.(ANL), Argonne, IL (United States), 2017.

[89] L. Xu, S. Qiu, B. Yuan, J. Jiang, C. Renggli, S. Gan, K. Kara, G. Li, J. Liu, W. Wu, et al. Stochastic gradient descent without full data shuffle. *arXiv preprint arXiv:2206.05830*, 2022.

[90] M. Yamazaki, A. Kasagi, A. Tabuchi, T. Honda, M. Miwa, N. Fukumoto, T. Tabaru, A. Ike, and K. Nakashima. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650*, 2019.

[91] C. Ying, S. Kumar, D. Chen, T. Wang, and Y. Cheng. Image classification at supercomputer scale. *arXiv preprint arXiv:1811.06992*, 2018.

[92] A. B. Yoo, M. A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

[93] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves, et al. Amrex: a framework for block-structured adaptive mesh refinement. *The Journal of Open Source Software*, 4(37):1370, 2019.