Cluster Computing (2025)28:613
https://doi.org/10.1007/s10586-025-05283-3

Deep learning data handling: exploring file formats and access strategies

Edixon Parraga[1] · Betzabeth Leon[1] · Sandra Mendez[2] · Dolores Rexachs[1] · Daniel Franco[1] · Emilio Luque[1]

Received: 13 September 2024 / Revised: 18 March 2025 / Accepted: 5 April 2025

Abstract

Accessing large volumes of data presents a significant challenge when finding the best strategies to manage the data efficiently. Deep learning applications require the processing of massive amounts of data, which implies a considerable access Input/Output (I/O) load on computer systems. During training, interaction with the I/O system intensifies as files are continuously accessed to read data sets. This persistent access could overload the file system, which, in turn, adversely impacts application performance and efficient storage system utilization. Several factors influence the I/O of these applications, and one of the most relevant is the variety of file formats in which datasets can be stored. The choice of file format depends on the use case, as each format defines how information is stored. Some file formats have features that promote efficient access to datasets during the training phase, which can improve the performance of deep learning applications. Likewise, it is also important that the format adapts to the context, in this case, to an HPC system with a parallel file system. We will propose an image preprocessing method for cases where performance improves with parallel file access. This method will transform image data sets from their original JPEG format to the more efficient HDF5 format. Thus, our research will focus on the importance of understanding the mode of data access, spatial and temporal patterns, and the level of parallelism in file access to determine whether it is advisable to change the storage format.

Keywords File format · Deep learning · Scalability · High-performance computing · Parallel input/output

✉ Edixon Parraga
edixon.parraga@uab.es

Betzabeth Leon
betzabeth.leon@uab.es

Sandra Mendez
sandra.mendez@bsc.es

Dolores Rexachs
dolores.rexachs@uab.es

Daniel Franco
daniel.franco@uab.es

Emilio Luque
emilio.luque@uab.es

[1] Computer Architecture and Operating Systems Department, Universitat Autònoma de Barcelona (UAB), Edifici Q. Escola d'Enginyeria - Carrer de les Sitges, 08193 Bellaterra, Barcelona, Spain

[2] Computer Sciences Department, Barcelona Supercomputing Center (BSC), Plaza Eusebi Güell, 1-3, 08034 Barcelona, Barcelona, Spain

1 Introduction

Parallel input/output (I/O) has been a hot topic among the High-Performance Computing (HPC) community for decades, driven by the perennial gap between processing and data access speeds and by increases in the scale of HPC architectures and, therefore, the I/O requirements of the applications. According to [1], achieving efficient parallel I/O in High-Performance Computing applications is challenging due to the complex interdependencies between multiple layers of I/O middleware and storage hardware. Each I/O middleware layer offers a set of tunable parameters. However, configuring these parameters for the best possible I/O performance depends on several factors, including the application, file system, storage hardware, problem size, and number of processors.

Parallel Deep learning (PDL) applications use large volumes of data, introducing large I/O workloads to HPC systems. High frequency and long duration of file system access could negatively impact response time. In the training stage of a neural network, it is often necessary to access thousands or millions of data or files with various characteristics in terms of size, number of files, structure, and data distribution, among others. This constant interaction could be an I/O intensive problem, requiring I/O storage and management requirements commensurate with the frequency and volume of information handled.

Handling large volumes of data is among the essential tasks of large supercomputing centers. The storage system of this data refers to the integrated hardware and software systems used to capture, manage, and classify it. The importance of its storage revolves around guaranteeing access to information and ensuring its permanence: to locate information quickly, process it to make better decisions, and use it to make predictions. When the data is created, it is generated in a format that will meet the expectations for the initial use for which they were intended. But, if you want to process that same data in another way, such as to train a neural network and be able to make predictions, the original format might not be the most appropriate.

Choosing the right data format is important for optimizing an application’s performance, as some formats are more efficient in terms of read/write speed and resource usage than others. In the context of AI algorithm training, where various data types are handled, native formats from deep learning libraries such as TensorFlow and PyTorch are expected to be encountered. This may require data conversion to these formats that use tensors as the main structure. The choice of data format depends on the specific task, such as classification, regression, image or text processing. It is essential to adequately preprocess the data for it to be compatible with the deep learning model as well as for it to consider the storage system and the possibility of parallel processing. This is especially relevant when working with deep learning applications in HPC systems, where efficiency in data storage, access, and parallel processing plays a key role.

Image datasets typically contain millions of small files distributed in thousands of directories. These data, in different formats, were not initially created for processing; the vast majority require preprocessing to be ready for later use. Figure 1 shows the data’s stages; after generation and acquisition, the data must be preprocessed to remove inconsistencies, cleaned and adapted to the intended use. The data could then be stored, or preprocessing could be performed to reformat it and make it easier to access for further analysis, visualization, and display.

Choosing the right file format can impact the storage system (bandwidth, read and write time), the data structure of the file (layout of the file structure), the access mode of processes (single, shared), and compression support,

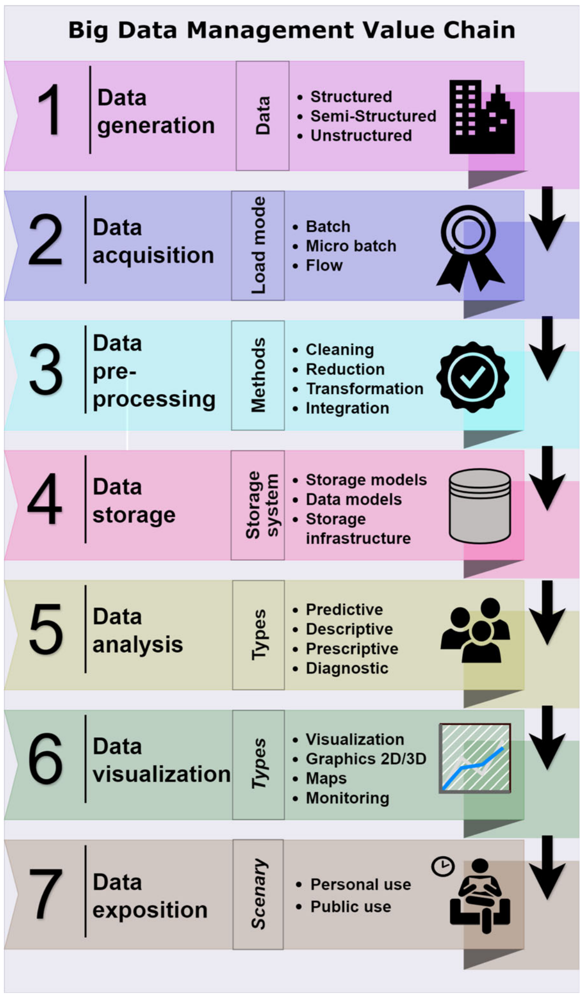


Fig. 1 Big Data Management Value Chain (Adapted from [2])

among others. The way information is handled in parallel DL applications needs to be different [2] and adapted to the needs of variability, randomness, frequency, and repetitive use that these applications naturally have.

Therefore, the file format could impact the training stage when the dataset is read since opening and reading millions of small files could affect the application's performance. The access mode to the dataset with different types of file formats could be another element of impact on the execution of the application, which is relevant if any of these types of file formats can be accessed in parallel. Therefore, depending on the case, the type of file format which could be more suitable should be evaluated and whether it is advisable to do some preprocessing, which requires time and resources to change the file format before reading the dataset.

Consequently, this article proposes an image preprocessing method for parallel file access. To do this, we will apply our method to the Tiny ImageNet-200 data set, originally in JPEG format, and we will transform it to HDF5 format in an HPC environment with a parallel file system. We will examine how this transformation affects I/O behavior and application performance, thereby providing valuable information for making preprocessing recommendations and optimizing application performance. These findings can guide the choice of the most appropriate format or determine whether it is beneficial to change the file format before starting application execution or processing.

This paper is structured as follows: Sect. 2 refers to Background and Related Work. Section 3 addresses elements that can impact the I/O behavior of a PDL application. Section 4 proposes a data preprocessing method to enable parallel file access using the HDF5 format. Section 5 presents experimental results on file format transformation, I/O behavior, and parallel application performance. Chapter 6 establishes some considerations for selecting the file format based on the results obtained. Finally, in Sect. 7, we present our conclusions.

2 Background and related work

Deep learning (DL) frameworks increasingly use HPC systems to speed up training large and complex neural networks. During a training process, the connection weights in a network are iteratively adjusted to minimize a loss function that reflects the error of a network. This process involves feeding the network with many training samples, which implies computationally intensive matrix multiplications. Like many HPC applications, I/O is one of the main performance bottlenecks, as the training process requires ingesting many samples. Training datasets typically contain several Gigabytes and up to several Terabytes of data. Thus, it is important to characterize the impact of I/O on training performance [3] to minimize the impact by establishing strategies that improve the I/O phase of the applications. The datasets used by deep learning applications are represented by a wide variety of formats such as text, binary, images, audio, video, and compressed, as well as exclusive formats for one type of data and formats such as HDF5, which allow data of different nature to be stored in the same file and relate them to each other.

File formats are how files are stored and define how information is organized and encoded in a computer file. Both in the scientific field and in artificial intelligence, it is necessary to use different formats especially adapted to each data type that will be stored. The main problem lies in standardization and interoperability since, although there are numerous information storage standards, not all behave the same when processing this data in the training phase.

Although each stack layer (see Fig. 2) offers adjustable parameters to optimize performance, application developers require more guidance on how these parameters interact and impact the overall I/O performance. As noted by the authors in [4], an I/O stack consists of multiple layers, including high-level I/O libraries, I/O middleware, and parallel file systems, each with numerous parameters to consider. This study focuses explicitly on file format and its influence on I/O behavior and, consequently, on the performance of a DL application. Parameters such as access type, file size, and the number of files per process will be analyzed, as these factors can significantly impact the application's I/O behavior.

All applications need to read or write data, which is an element that must be considered when improving the performance of DL applications. Consequently, given the great variety of data this type of application handles, the file formats used in the data sets are also varied. Each

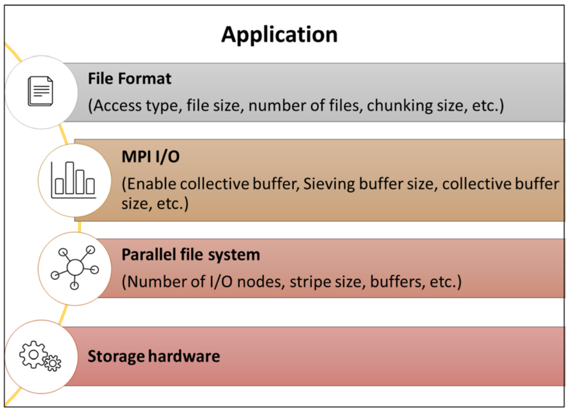

Application
File Format
(Access type, file size, number of files, chunking size, etc.)
MPI I/O
(Enable collective buffer, Sieving buffer size, collective buffer size, etc.)
Parallel file system
(Number of I/O nodes, stripe size, buffers, etc.)
Storage hardware

Fig. 2 Some adjustable parameters of the Parallel I/O Software Stack (Adapted from [5])

format has specific characteristics that can affect the performance of DL applications. Two formats used in this context for image recognition are JPEG and HDF5, on which this study will focus.

Hierarchical data format version 5 (HDF5):

HDF5 datasets, as defined in [6], serve as organized containers for raw data values. The distinctive feature of the HDF5 format lies in its ability to incorporate metadata, rendering it self-descriptive. This built-in metadata facilitates extracting information about the dataset's content from within the file, eliminating the need for external sources.

HDF5 objects, including datatypes, dataspaces, properties, and, optionally, attributes, play a pivotal role in describing a dataset. Datatypes comprehensively describe individual data elements within the dataset, offering essential details for data conversion to and from that specific data type. Dataspace defines the arrangement and layout of data elements within the dataset. Properties encapsulate distinct characteristics or attributes of an HDF5 object. Attributes, on the other hand, are optional and can be linked to HDF5 objects.

Main features of the HDF5 format:

- Hierarchical structure: HDF5 organizes data in a hierarchical tree structure, similar to a computer file system. Data is stored in 'groups' and 'datasets.' Groups are similar to folders, and datasets are similar to data files.
- Support for large volumes of data: It is designed to handle massive amounts of data.
- Support for multiple data types: Supports various data types, including integers, floats, strings, and complex structures such as multidimensional arrays, images, and tables.
- Efficient and fast access: Provides efficient and fast access to data, allowing operations such as reading and writing subsets of data without loading the entire set into memory.
- Portability: HDF5 files can be used on different operating systems and hardware architectures without conversion.
- Integrated metadata: Allows metadata to be stored along with the data, making it easy to describe the data and track its provenance.
- Application and programming language support: It supports various programming languages.

JPEG (Joint Photographic Experts Group):

JPEG is a widely used image compression format, especially for photographs and color graphics [7, 8]. Main characteristics of the JPEG format:

- Lossy compression: The JPEG format uses a lossy compression algorithm, meaning that some of the original information is lost when an image is compressed.
- Smaller file size: Due to their compressibility, JPEG images are usually much smaller in file size compared to other formats such as PNG or BMP.
- Support for millions of colors: JPEG can handle up to 16.7 million colors (24-bit).
- No support for transparency or layers: Unlike formats such as PNG, JPEG does not support transparency or the ability to store multiple layers in an image.
- Broad compatibility: JPEG is one of the most widely supported image formats, compatible with almost all devices, operating systems, and image editing software.

JPEG offers several compression modes: Hierarchical compression allows multiple resolutions in one image for quick thumbnail viewing. *Progressive compression:* Allows gradual display of the image as it downloads, improving the experience on slow connections. *Sequential compression:* Compresses and saves the image in a single pass, prioritizing processing speed. Lossless compression: Reduces file size without losing quality, ideal for images where precision is essential [9].

File access mode

The access modes for a file determine how users and processes can interact with it. Two of these file access modes are independent and shared.

Independent access

Each file format has distinctive characteristics, particularly in how files are accessed. In this context, Fig. 3 illustrates the concept of independent access, where each process autonomously interacts with a file. This mode of independent access is exemplified in this work for the JPEG file format. The term"independent"implies that each process can invoke functions without depending on or coordinating with other processes. There are no limitations on the number of calls a process can execute, and they do not need to synchronize their calls with those made by other processes [10].

Shared access

Shared access means that multiple processes are accessing the same file (Fig. 4a). This access can be distributed, where each process accesses different parts of the file, or full access, where each process accesses the entire file.

Distributed sharing access:

Shared access mode involves simultaneous access by multiple processes to the same file. If accessed in a distributed manner, they can read or write to different file

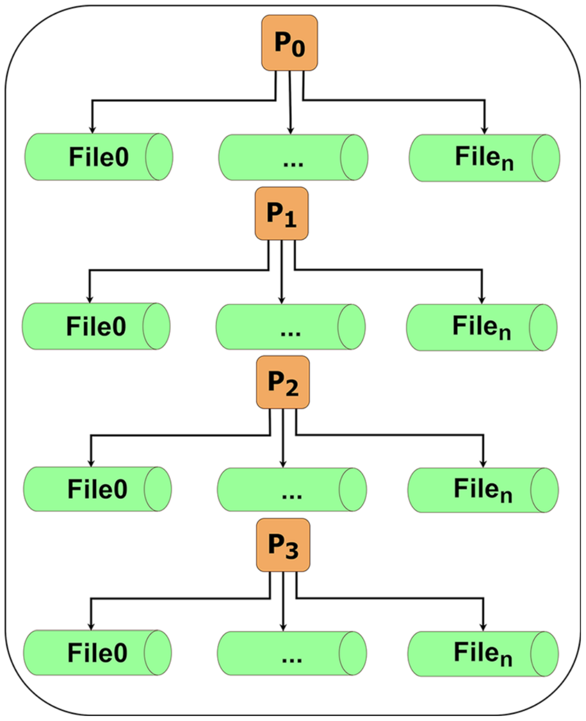


Fig. 3 Independent Access

sections simultaneously. This collaborative approach offers several advantages and can improve overall system performance. Each process can operate simultaneously on different segments of the same file, allowing efficient workload distribution and speeding up read and write operations. Unlike duplicating the file for each process, shared access eliminates unnecessary data replication, which conserves disk space and reduces memory usage.

In the case of HDF5, it can support this access mode when necessary, allowing all processes to access the same file in a shared manner by dividing it into segments allocated to each participating process. Consequently, with this file format, achieving parallelism when accessing data within a file becomes feasible, as demonstrated in Fig. 4b.

Full sharing access:

In this scenario, all processes simultaneously access the same HDF5 file and read its contents. Here, all processes access the entire file, not just a specific part, implying that they all read the same complete data set. Figure 4c illustrates this approach, showing how each process has full access to the HDF5 file. This can be advantageous when all processes must use the same data for consistent calculations or analysis. However, it can also increase I/O demand, which impacts performance if not managed properly.

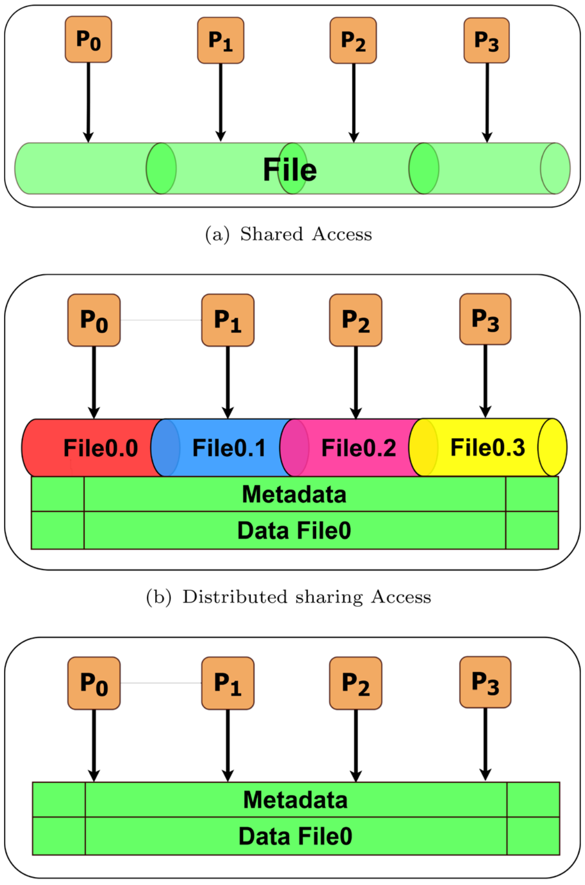


Fig. 4 Shared file access mode

2.1 Related work

Regarding related literature, some studies are mainly based on high-level libraries and the various HDF5 utilities. Concerning HDF5, in [1], the authors showed that automatic tuning of parallel I/O parameters in HDF5 applications could improve I/O performance without practical optimization or code changes. In [11], the authors proposed a new specification called Exdir, which puts the HDF5 abstractions on top of a hierarchical directory structure. It has similar flexibility to HDF5 in a simple form, human-readable metadata, and the applicability of established tools. In [12], the authors presented an approach to integrate node-local storage, such as transparent caching or staging layers, into a high-level parallel I/O library without placing the burden of managing these layers on users.

Regarding high-level libraries and file formats, in [13], the authors presented an asynchronous I/O framework that supports all types of I/O operations, manages data dependencies transparently and automatically, as well as providing implicit and explicit modes for application flexibility and error information recovery. In this work, the authors implemented the techniques in HDF5.

Likewise, in a report presented by MLPerf [14], the authors indicated that an important step to maximize performance was to use an optimized HDF5 file format for the data set. With this, it was possible to get maximum data loading performance. These investigations address HDF5 in their work, one of today's most widely used file formats. It is used in HPC and applications for handling large volumes of information, such as deep learning and artificial intelligence.

In other related literature, several studies have proposed methods to reduce the training time of deep learning applications. In [15], the authors focused on identifying the file I/O factors that affect the performance of Intel-Caffe and on evaluating performance in a container-based environment. This study determined that the DL training process in a container-based environment has minimal overhead; likewise, the authors found that modifying the shuffle algorithm to utilize the page cache can fully alleviate the bottleneck in file I/O.

Likewise, in [16], the authors presented a transient runtime file system that optimizes DL I/O in existing hardware/software stacks. This job uses local storage to enable efficient and scalable DL training. In [17], the authors analyzed LMDB (Lightning Memory-Mapped Database), a widely used I/O subsystem of deep learning frameworks, to understand the causes of I/O inefficiency.

In [18], the authors presented a distributed deep learning framework for a heterogeneous multi-GPU cluster to improve overall resource utilization without sacrificing training accuracy. This design was implemented using MPI and Tensorflow and trained ResNet 50 on the ImageNet dataset. In [1], the authors worked with a benchmark that identifies sets of parameters with good performance for a given system and problem size. I/O parameters are specified in an XML configuration file that is read by the H5 Tuner library. Through this library, the authors intercepted the HDF5 function calls of the application to adjust the parameters based on the configuration file's content to perform automatic tuning.

These studies approach optimizing I/O operations in deep learning applications from distinct angles to mitigate their impact. They delve into various aspects, including the HDF5 file format, file systems, the behavior of different frameworks, workload management, and performance parameters, among others. Collectively, these studies contribute to identifying multiple components that play a role in the I/O processes of deep learning applications.

3 I/O behavior of a DL application

The I/O behavior of parallel deep learning applications (spatial and temporal patterns of I/O operations) is subject to several factors, including the internal configuration of the application, its underlying hardware, and the characteristics of the file system in use. Specifically, we focus on two key aspects, which are the file format employed by the deep learning application and the parallel access mode of the processes, as illustrated in Fig. 5.

The file format, its structural organization, compression techniques, and other file-related attributes are all pivotal in shaping the spatial and temporal patterns of I/O operations. Consequently, these factors substantially influence the application's overall I/O behavior.

In the study [19], the authors present a comprehensive classification of I/O access patterns. I/O operations can typically be categorized as read-only, write-only, or a combination of both, encompassing read and write actions. The size of these I/O requests can further be characterized as fixed, variable, small, medium, or large.

The spatial pattern of an application is best represented by the sequence of file locations accessed during its operation. These operations may manifest themselves as contiguous, non-contiguous (stride/random), or a blend of both. Non-contiguous access patterns involve interruptions or gaps in file access, which can be fixed or variable. Alternatively, some access patterns exhibit decreasing (negative) step sizes, while others lack a regular pattern featuring random steps. Small requests can lead to performance bottlenecks, as frequent disk accesses for minimal data and high I/O latency can impact overall efficiency.

The temporal pattern characterizes how applications behave over time. It distinguishes between applications with repetitive behavior, where loops or functions with loops generate recurrent I/O requests and those with single-occurrence patterns. The temporal pattern captures the regularity in an application's I/O bursts, which may occur periodically at fixed intervals or sporadically at random moments.

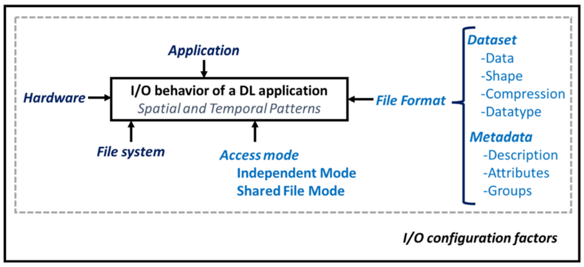


Fig. 5 Elements that impact the I/O behavior of a PDL application

The significance of categorizing these access patterns lies in the potential to implement strategies that mitigate the impact of I/O operations on applications. For instance, in repetitive I/O access patterns, techniques like caching and prefetching can effectively mask the associated access latency. By recognizing these repetitive behaviors, cached data can be retained for extended periods or accessed to maximize its utility before accommodating new data. Prefetching capitalizes on this regularity by leveraging prior patterns to predict future I/O access offsets. In practical terms, understanding temporal regularities can guide strategies to initiate prefetch requests proactively, ensuring that prefetched data enters the cache precisely when needed. Additionally, regarding request size, aggregating numerous small accesses into larger, contiguous requests effectively reduces the number of disk seeks and enhances overall I/O performance.

Configuration factors

When we run a deep learning (DL) application, we encounter several configurable factors that can influence how a file is read and written. These elements are important in DL application input and output (I/O) management and significantly impact performance. These factors are detailed in Table 1 and are of utmost importance in planning and optimizing I/O operations in a deep learning application. It is imperative to consider these factors depending on the file format selected to ensure efficient performance.

The total number of processes (p) influences the choice of file format, and it is essential to weigh the efficiency of formats capable of parallel reading and writing, especially when dealing with a substantial volume of processes accessing a dataset concurrently. The size of each file (sf) plays a pivotal role in format selection. In cases where files are huge, it may prove more efficient to divide them into smaller, manageable chunks or opt for formats that facilitate data storage in batches.

In addition, in situations where the dataset comprises a substantial number of files (nf), adopting a format that efficiently organizes these files, such as through categorization and subcategorization within directories, can be particularly advantageous. Moreover, when numerous processes attempt simultaneous access to the same file (pf), selecting a format that facilitates concurrent reading without causing blockages is critical.

The maximum number of files a process intends to access (mf) can impact the choice of file format. For processes necessitating access to numerous small files, opting for a format supporting data batching can yield significant gains. Lastly, elements such as the total number of nodes (n) and the number of processes per node (ppn) are important to distribute data efficiently between these nodes to achieve optimal performance. Moreover, the total number of files (nf) that comprise the dataset's structure and organization can significantly impact the management and efficiency of data search in a large data set.

4 Proposed image preprocessing method to allow parallel file access

Data preprocessing is important in machine learning and deep learning. It involves exploring the data to verify its validity, correcting and removing inconsistent information, replacing or eliminating null or invalid data types, adjusting data scales, removing unnecessary data, and adding new fields. Additional preprocessing beyond data cleaning can optimize their reading during training when the dataset includes images. This can be achieved by adjusting the image format to improve performance through parallel I/O accesses.

Figure 6 illustrates the proposed image preprocessing method to facilitate simultaneous file access in high-performance environments. This method consists of three main phases. The first phase involves preparing the dataset, which includes organizing, selecting, and structuring the information to convert it into a format more suitable for further processing. The second phase focuses on image preprocessing, where specialized code is run to validate the conversion and ensure data integrity and accessibility. Finally, the preprocessed data is utilized during the third training phase, optimizing parallel data access and loading. This method aims to enhance efficiency in handling large volumes of image data by utilizing a storage and access

Table 1 Key Configuration Factors Influencing File Format Selection

Acronym	Category	Description
p	Mapping	Total number of processes
n		Total number of nodes
ppn		Number of processes per node
sf	File	Size of each file
nf		Total number of files in the dataset
pf		Number of processes used to open a file
mf		Maximum number of files that will be accessed by a process

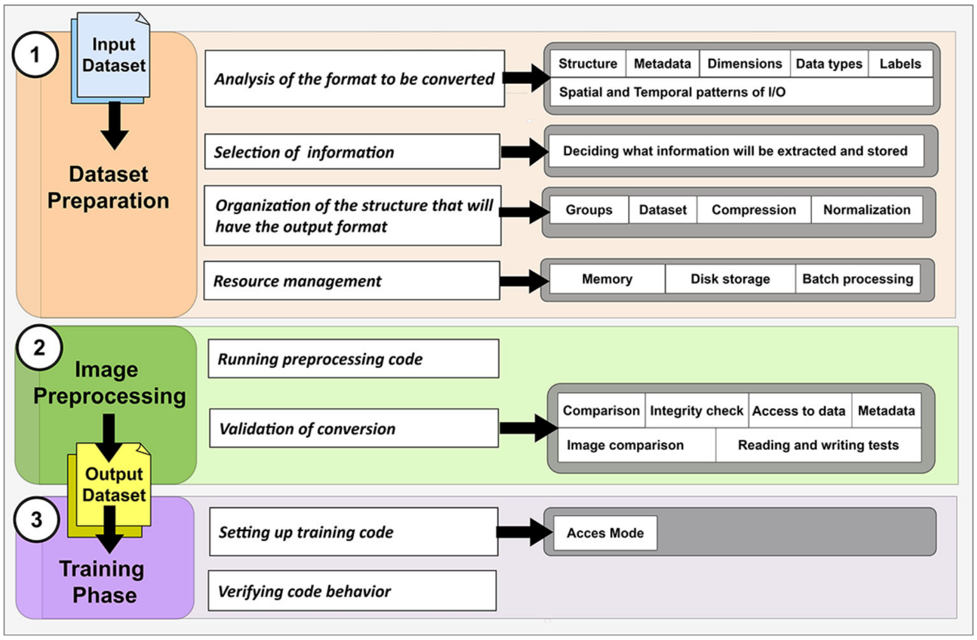


Fig. 6 Image preprocessing method to allow file access in parallel

format that supports more effective simultaneous input/output operations.

As a case study, we will explore the transformation of a dataset from JPEG format to HDF5 format. The HDF5 format offers numerous benefits due to its adaptive input/output operations capability. It accommodates large and heterogeneous datasets, presenting information in a structured and comprehensive manner with metadata, and allows simultaneous access to data files. In a later section, we will use the Tiny ImageNet-200 [20] dataset to demonstrate the application of the preprocessing steps described in the method. Although the Tiny ImageNet-200 dataset is relatively small, its selection responds to practical constraints in resource usage, including reduced execution times and lower energy consumption, which optimize both the experimental process and its environmental impact. Furthermore, it provides an effective means to study file format transformations, whose principles can be extrapolated to larger datasets.

In a previous study [21], we analyzed temporal and spatial I/O patterns in HDF5 datasets, showing the applicability of these principles to a broader range of scenarios. This earlier work laid the groundwork for understanding the scalability and efficiency of the HDF5 format, which we further explore in this study. Each step of the proposed method is explained in more detail below.

4.1 Dataset preparation

A thorough analysis of the data structure embedded in the JPG file format is vital at this stage. This analysis covers various aspects such as dimensions, data types, tags, metadata, and other relevant details. A spatial and temporal analysis of the I/O patterns generated by this format is also performed. A meticulous selection process is essential to determine what information should be extracted and stored in HDF5 format. The structure must be precisely designed to ensure full compatibility with the HDF5 format. Careful decisions regarding its encoding within the HDF5 file are critical, mainly when the data includes tags. In addition, if additional information from the dataset is to be retained, it is necessary to store this data appropriately as metadata within the HDF5 file.

Algorithm 1 Case Study: Transforming JPEG to HDF5

```
1: Configure Parameters:
2:     data_format: Determines the desired output format.
3:     folder_path: Path to the folder containing the images.
4:     output_dir: Directory where the transformed files will be saved.
5:     filename: Output file name based on data_format.
6: Upload Images
7: Check Images
8: Transform and Save Images (Example case of transforming JPEG to HDF5):
9:     Initialize HDF5 file in output directory with optional compression
10: for each subfolder in folder_path do
11:         Create group in output_hdf5_file based on subfolder name
12:     for each image file in subfolder do
          Try:
13:                 Convert image to NumPy array
14:                 Save image to the dataset in the corresponding group
          Except:
15:                 Handle any errors (e.g., log the error, skip the image)
16:     end for
17: end for
18:     Close output_hdf5_file properly
19:     Print or Log processing time and size of the HDF5 file
```

To construct the HDF5 file effectively, the data should be systematically organized into groups and datasets within the HDF5 framework. Groups can represent categories or classes, while datasets house the content. Additionally, careful consideration should be given to the potential implementation of compression methods tailored to specific requirements, as the choice of compression may have implications for the data organization within the file. When the data encompasses varying value ranges, it's important to undertake appropriate normalization and scaling measures in order to maintain consistency throughout the conversion process.

Similarly, it's imperative to incorporate efficient resource management strategies within the script employed for format transformation. This includes reasonable memory allocation during transformation to preempt resource depletion errors. When dealing with extensive datasets during format transformation, memory-related issues can arise due to accumulating images and data in memory throughout the execution.

Given the utilization of large datasets and memory-intensive processes, efficient resource management becomes paramount to avert potential problems stemming from resource exhaustion. Among the strategies that can be used to reduce memory usage during format transformation are the following [22, 23]:

- **Disk storage:** This strategy consists of storing images on disk instead of loading them into memory. This means that images are loaded into memory only when necessary, reducing the pressure on RAM. When an image is needed, it is loaded from the disk and processed.
- **Batch processing:** Images can be processed in smaller batches and then added to the HDF5 file, avoiding simultaneously loading all images into memory.
- **Compression techniques:** Images can be compressed before storing in the HDF5 file.
- **Use of generators:** It is possible to create a generator that reads and processes the images as needed. This helps keep memory requirements low by loading one image at a time into memory.

Each of these strategies presents its own set of advantages and drawbacks. Disk storage and batch processing alleviate memory constraints but may extend data access times. Compression effectively conserves disk space but may necessitate extra time for decompression. Using generators proves memory-efficient but can lead to slower data access rates.

All these strategies offer viable solutions for managing the transformation of extensive datasets. The unique requirements of each case should guide their selection.

4.2 Image preprocessing

After analyzing the previous format pattern by studying the I/O behavior, preprocessing is carried out using a code specifically designed for this task. Algorithm 1 presents, in general, the steps required to transform images from JPEG to HDF5 format. First, the parameters corresponding to the input and output directories and files are configured. Next, the images are loaded and it is verified that everything is in order. Once the images have been validated, the format transformation is carried out, in this specific case, from JPEG to HDF5.

To carry out the transformation, an HDF5 file is created and, for each subfolder, the necessary groups and datasets are generated. Each image is stored in its respective dataset within the corresponding group. Error handling is included to ensure the robustness of the process. Finally, the HDF5 file is correctly closed, and the logs are recorded, along with information on the time used and the size of the generated HDF5 file.

Depending on the size of the JPEG data set, this process may require considerable time. Therefore, this preprocessing step is strongly recommended, especially when the training phase is extensive. The dataset format transition can potentially offset the time spent on preprocessing by reducing the overall duration of the training process.

It is also essential to consider storage resources, as the resultant file generated from this transformation may occupy a substantial amount of disk space. Numerous temporary files may be generated depending on the resource management strategy employed, further impacting disk space availability.

Following the completion of data preprocessing, which converts it into the HDF5 format, validating the accuracy of the data transformation process becomes imperative. Ensuring the absence of important data loss during the conversion is paramount. Verifying the structure of the created HDF5 file is equally important, as it must match the loading and reading function of the training code used. This validation is pivotal since subsequent configurations in the training code will rely on this structure for efficient dataset loading and reading.

Validating an HDF5 file generated from a JPEG to HDF5 format transformation involves verifying proper data retention and correct file structure. To perform this validation, several tests can be performed:

- **Integrity check:** The integrity of the HDF5 file is checked for possible corruption issues.
- **Data access:** The HDF5 file is opened to access HDF5 data and methods, thus ensuring proper data preservation. This can be verified by extracting a random subset of the data and comparing it to the original data in JPEG format.
- **Metadata comparison:** This verifies that the relevant metadata has been correctly preserved in the HDF5 file. This includes information about dimensions, data types, tags, and other details mentioned in the transformation.
- **Read and write testing:** Read and write tests are performed on the HDF5 file to confirm that the data can be retrieved and modified as required.
- **Comparison of images in both formats:** Random image readings can be made in the HDF5 file to compare them with those of the original JPEG file.

This validation is essential to ensure the generated HDF5 file is accurate and suitable for future use.

4.3 Training phase

After validating the dataset in its transformed format, it is essential to ensure that the training code is adequately configured to load data that conforms to the new file format and the assigned shared access method. The user is responsible for adapting the data loading in their code to align with this new format. Once this configuration has been successfully implemented, subsequent steps include reading the data and starting the training phase.

Conducting code behavior verification with the dataset in the new format is also advisable during this phase. This verification process should include comparing the code's performance with the new format and its behavior when working with the original format.

Our method can be fully automated by developing tools that dynamically adjust critical parameters during format transformation. These parameters include:

- Structure of the output file (HDF5):
  - Number of groups and datasets: Determined by the number of folders and the images per folder in the original dataset.
  - Chunk size: Can be automatically adjusted to optimize parallel access depending on the file system characteristics and the observed access patterns.
  - Compression: Configuration of compression filters (such as Gzip) based on storage requirements and runtime decompression capabilities.
- Resource Management: Implementation of strategies such as batch processing, pre-compression of images, and the use of generators to efficiently handle large datasets without exceeding the available memory resources.
- Automated Validation: Inclusion of integrated checks to ensure data integrity and correspondence with the original data.

This automated approach would enable handling larger-scale datasets with greater structural diversity and facilitate the process's replicability across different machine-learning scenarios. Additionally, these tools could be extended to include recommendations based on analyzing access patterns and optimizing parameterization without manual intervention.

Table 2 Neural Network Layer Configuration

Layer Type	Details
Convolutional layer 1	Filters: 32
	Filter Size: 3x3
	Activation function: Rectified Linear Unit (ReLU)
	Input: 64x64 pixels, 3 channels (RGB)
Max-Pooling layer 1	Pooling size: 2x2
Convolutional layer 2	Filters: 64
	Filter Size: 3x3
	Activation function: ReLU
Max-Pooling layer 2	Pooling Size: 2x2
Convolutional layer 3	Filters: 64
	Filter Size: 3x3
	Activation function: ReLU
Flatten layer	This layer converts the two-dimensional outputs of the last convolutional layer i nto a one-dimensional vector, which can be fed to the subsequent dense layers
Dense layer 1	Units: 64
	Activation Function: ReLU
Output dense layer	Units: num_classes (variable representing the number of classes in the dataset)
	Activation Function: Softmax (for multi-class classification)

5 Experimental results

This section presents the experimental results and their respective analyses. It involves applying the image pre-processing method introduced in the previous section. Here, we showcase the performance achieved through the format change. Before delving into the experimental results, we will describe the experimental environment used.

Experimental environment:

The experimental environment was on the FinisTerrae III, which is made up of 354 nodes that in total include 708 cutting-edge Intel Xeon Ice Lake 8352Y processors with 32 cores at 2.2 GHz, bringing FinisTerrae III to 22,656 processing cores. Eighty nodes incorporate 144 GPU math accelerators: Nvidia A100 and Nvidia T4 GPU, giving it a total computing power of 4 PetaFLOPS. The file system used was LUSTRE. The default striping is 1, which means that, by default, only 1 OST (Object Storage Target) is used for each file, regardless of its size. The size of a stripe is 1MB. Because the generated HDF5 file is approximately 1.2 GiB, the default configuration will be used, which is the system recommendation.

Instrumentation tool:

We use Darshan 3.4.5, a scalable HPC I/O characterization tool, as a monitoring tool. Darshan is designed to accurately capture the application's I/O behavior, including properties such as access patterns within files, with minimal overhead [24].

Structure of the neural network used for testing:

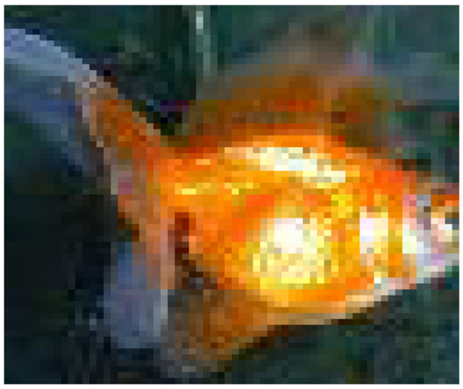
To test the performance of reading the dataset during the training stage and to compare the JPEG and HDF5 formats, a convolutional neural network (CNN) has been employed [25] using the TensorFlow library in Python, with support for distributed training using Horovod [26]. The structure of the neural network used is described in Table 2.

Optimization and Training

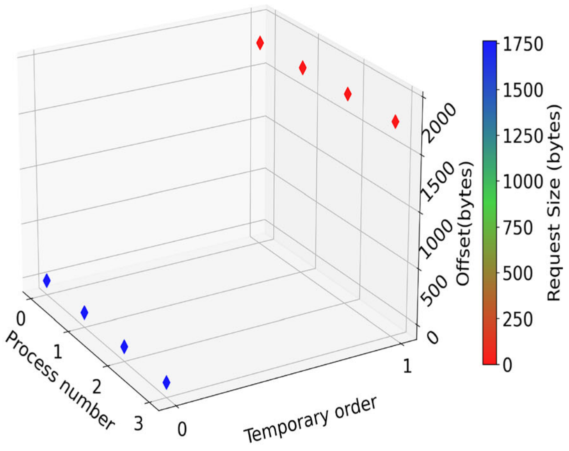
The optimizer is Adam [27], tuned for distributed training using the Horovod implementation. The loss is sparse_categorical_crossentropy, suitable for classification problems where the labels are integers (categories). The model is trained for 5, 10, and 15 epochs, and global variables are synchronized at each epoch using Horovod's Broadcast Global Variables Callback.

5.1 Impact of format change on the performance of a DL application

In this section, an analysis of the format change process will be carried out on the dataset Tiny Imagenet-200 [20]. This dataset is comprised of 100,000 color images reduced to a resolution of 64x64 pixels in JPEG format, distributed in 200 classes in which each class has 500 images per class. It also has 10,000 evaluation images and 10,000 test



(a) Image 215 of class n01443537 (Tiny ImageNet 200 (Le & Yang, 2015) [28])



(b) Spatial and Temporal Pattern

Fig. 7 I/O behavior of a Tiny ImageNet-200 dataset file JPEG format

images. For experimentation purposes, we will only use the train directory.

The spatial and temporal patterns of input/output (I/O) operations on this data set in its original format (JPEG) will be examined. Subsequently, our image preprocessing method will transform this data set to HDF5 format. A detailed analysis of the temporal and spatial patterns of I/O operations in this new HDF5 format will also be carried out during this conversion process. Finally, we will perform a performance comparison using the same application and dataset but in two different data formats: JPEG and HDF5. This analysis will allow us to evaluate and compare the application's performance when working with these two different data formats.

5.1.1 Dataset preparation

Analysis of the format to be converted:

In the case of the Tiny ImageNet-200 dataset with JPEG format consisting of a large number of files (100,000 images), a focused analysis will be carried out on a single image file. This is because the I/O behavior of the other images follows a similar pattern. Therefore, by examining a single image file, we can properly understand and represent the overall I/O behavior across the entire data set. Image 215 of class n01443537 has been taken as an example, shown in Fig. 7a.

This image has a size of 1720 bytes, which translates into a spatial access pattern comprising only two read operations. First, access is performed at offset 0 with a size of 1720 bytes (as shown in the blue dot in the graph 7b), which is equivalent to the total size of the image, implying

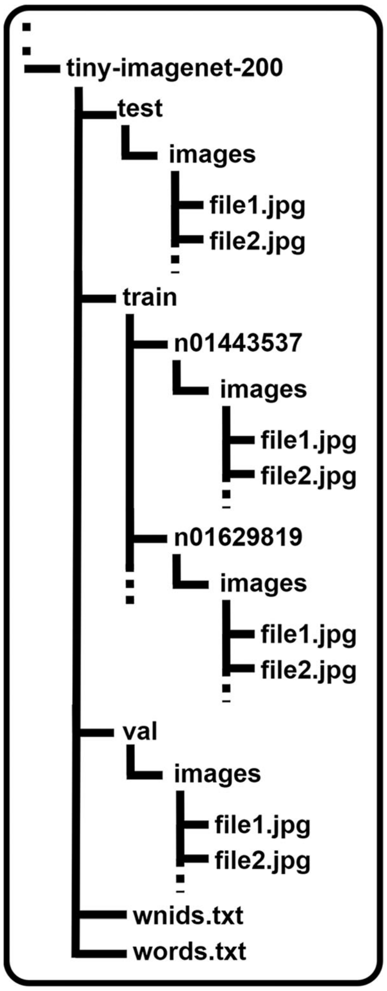


Fig. 8 Directory structure of the Tiny ImageNet-200 dataset

Fig. 9 Structure of the Tiny Imagenet-200 Dataset HDF5 generated

```
GROUP "/" {
    GROUP "train" {
        GROUP "n01443537" {
            DATASET "images" {
                DATA TYPE H5T_STD_UBLE
                DATASPACE SIMPLE { (500, 64, 64, 3) / (500, 64, 64, 3) }
                STORAGE_LAYOUT {
                    CONTIGUOUS
                    SIZE 6144000
                    OFFSET 141348232
                }
                FILTERS {
                    NONE
                }
                FILLVALUE {
                    FILL_TIME H5D_FILL_TIME_IFSET
                    VALUE H5D_FILL_VALUE_DEFAULT
                }
                ALLOCATION_TIME {
                    H5D_ALLOC_TIME_LATE
                }
            }
        }
    }
}
```

that it is read in its entirety. Then, a second access is performed at offset 1823 with a size of 0 (indicated by the red dot in the plot 7b).

As for the temporal pattern, only two accesses of different sizes are observed. This behavior is replicated in the rest of the images in the dataset. Due to the small size of these images in JPEG format, each process reads the files completely. Due to the independent access mode, each process opens the image, performs the two accesses, and then closes the image. Consequently, the total bytes read is obtained by multiplying the size of each image by the number of processes. For the example illustrated in Fig. 7, the total bytes read by the four processes is 6880 bytes (4 x 1720 bytes).

Selection of information:

As we saw in Section 4.1, the dataset is first prepared, and the structure of the image directory is identified. In this sense, Fig. 8 shows the structure, composed of three subdirectories at the first level test, in which there are 10,000 test images. The train subdirectory has 100,000 training images distributed in 200 folders, and in each folder, there are 500 images. The val subdirectory has 10,000 evaluation images. All of these folders, in turn, have a subdirectory with the name images, and inside these are the `images`.

In addition, there are two additional text files in which the meta information is located. In the `wnids.txt` file are the names of the 200 classes, and in the `words.txt` file are the names of the classes related to words that identify each class. In our case, only the train folder will be selected, with which the transformation from the JPEG format to HDF5 will be performed.

According to each case's specific needs, an HDF5 file must be designed to organize the images and tags into datasets grouped with their respective metadata. The size of the dataset will determine whether a strategy to manage memory resources efficiently is necessary to avoid loading all images into memory, which could lead to possible errors.

Organization of the structure that will have the output format:

In this phase, we executed the code to convert the data format. Specifically, we transformed the Tiny ImageNet-200 dataset from JPEG to HDF5 format. The code was designed to work exclusively with the training directory, so only this directory was considered for the conversion. The test and validation directories were excluded from this transformation, as the focus is solely on the training phase.

The structure of the HDF5 file generated with the preprocessing is shown in Fig. 9, where each folder from the original format (JPEG) was mapped to a group within the HDF5 file, and the images of each class were stored as datasets with associated metadata.

It is important to note that this transformation and the hierarchical structure of the HDF5 file would remain consistent as the number of images, folders, or classes increases. Each new folder from the original format would be mapped to a new group in the HDF5 file, and the images within each folder would be stored as datasets, replicating the same pattern observed in this analysis, regardless of the dataset's scale. This ensures that the logical organization of the HDF5 file faithfully reflects that of the original format and is scalable to larger volumes of data.

HDF5 File: The file is a container that organizes data into a hierarchy of groups and datasets.

- Root Group ("/"): The root group within the HDF5 file is/. This group, in turn, contains other groups and datasets.

- “train”Group: Inside the root group, there is a group called a train, which contains the training data. This group has subgroups that represent different categories or classes.
- Class Subgroups: Within the“train”group, each subgroup (e.g., n01443537, n01629819, etc.) represents a specific class. Each subgroup contains a dataset called images.
- “images”Dataset:

1. Data Type: The data is of type H5 T_STD_U8LE, meaning it is stored as unsigned 8-bit integers.
2. Dataspace: Each dataset has a dataspace of size (500, 64, 64, 3), indicating 500 images in each class, with dimensions of 64x64 pixels and three channels (RGB).
3. Storage Layout: The images are stored contiguously, meaning the data is stored in a continuous memory block.
4. Size: Each set of images occupies 6,144,000 bytes.
5. Offset: Each dataset has an OFFSET indicating where the dataset begins in the file.
6. Filters: No filters are applied to the data (no compression).
7. Fillvalue and Allocation Time: These properties indicate that values are filled only if needed.

Resource management:

At the Resource Management stage during data format conversion, it was decided not to apply advanced optimization strategies, such as batch processing, due to the nature of the dataset used. Specifically, we only worked with the train folder of the Tiny ImageNet-200 dataset, whose amount of data fits comfortably in the available memory. In addition, the size of the HDF5 file generated at the end of the process was only 1.2 GiB, which did not represent a challenge in terms of disk storage capacity. Therefore, it was not necessary to implement additional techniques for resource management, since memory and storage needs were amply covered.

5.1.2 Image preprocessing

Running preprocessing code and Validation of conversion

This data will be used to train and evaluate the machine learning model as an image classification model, where images are used as inputs, and labels will be used to monitor the training and evaluation process. The dataset was validated by verifying the file’s integrity and by the structure shown in Fig. 9 using code designed for this purpose.

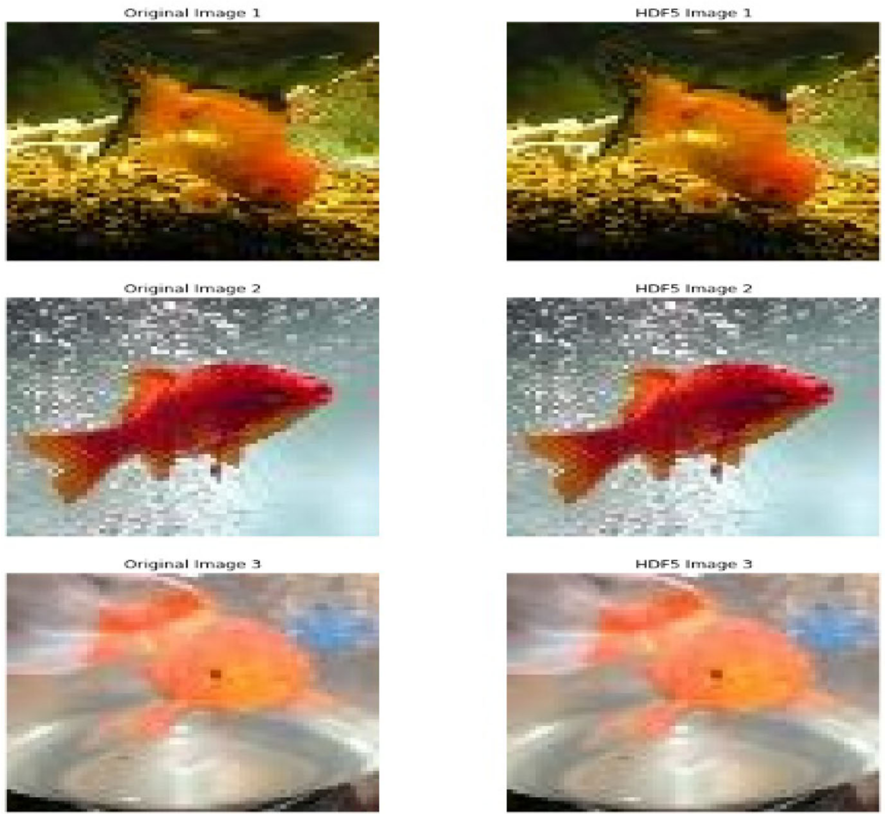


Fig. 10 Comparison of images from the HDF5 file with the original in JPEG (Tiny ImageNet 200 (Le & Yang, 2015) [28])

Table 3 Experiment mapping configuration

No. of nodes (n)	No. of processes (p)	No. of processes per node (ppn)	**Representation**
1	4	4	n1p4ppn4
2	4	2	n2p4ppn2
2	8	4	n2p8ppn4
2	16	8	n2p16ppn8

Subsets of data were randomly extracted from the images with their respective labels, which were compared with the original data in JPEG format, as shown in Fig. 10, where three original images in JPEG format were compared with three images randomly extracted from the HDF5 file. Likewise, it was verified that the metadata was stored correctly.

Following the code guidelines, the“train”directory selected for conversion was 402 MiB in size in JPEG format before transformation to HDF5. The format conversion from JPEG to HDF5 took 38.60 min, and the resulting file in HDF5 format reached 1.2 GiB.

The significant increase in dataset size when converting from JPEG to HDF5 is normal and expected. The main reason behind this increase is that the JPEG format is an image compression format that generally stores images in a highly compressed form, which reduces its file size. On the other hand, the HDF5 format is a data storage format that is generally more efficient at representing structured data and tags in their original form, which can considerably increase file size.

In addition, HDF5 stores additional metadata and structure not found in a JPEG file, contributing to the increase in size. If the original dataset contained tags, directory structure, and metadata, these will also be included in the HDF5 file, making it larger.

5.1.3 Training phase

After successfully validating the dataset in the new HDF5 format, we moved to the model training phase. We modified the same codebase previously used for the JPEG dataset to allow data to be loaded and read in HDF5 format datasets.

For the training phase, we designed experiments focusing on the configuration factors described in Sect. 3. These factors cover mapping and access modes for the HDF5 and JPEG datasets. Table 3 shows an overview of the specific mapping configuration used.

As for the number of epochs to verify the integrity of the file format change to HDF5, multiple epochs (5, 10, 15) were experimented with. For experiments on the impact of the format change on application I/O with different mapping, the results of 5 epochs are shown for all cases. This decision is because running many epochs in deep learning applications can be computationally expensive and time-consuming. This is because the model is fine-tuned to the training data in each epoch, which repeatedly loads and reads the same data from storage, resulting in high I/O traffic. This is why format switching is important. Since our main focus was to evaluate how file formats influence I/O, it was not necessary to use a large number of epochs to observe the influence of formats on file reads.

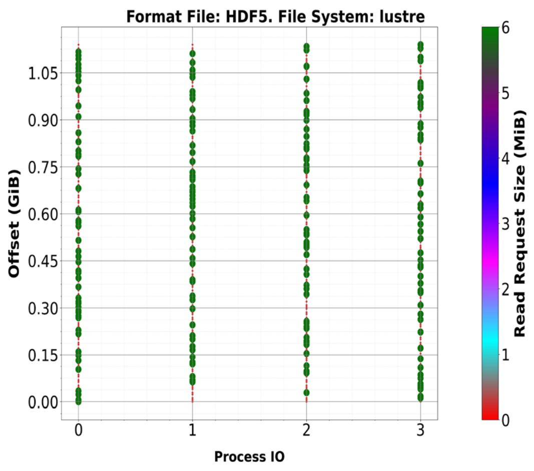


Fig. 11 HDF5 Distributed Sharing: Spatial Pattern

Furthermore, to identify the spatial and temporal pattern of I/O, we limited the experiments to a single epoch to focus clearly on the interaction between the model and the data files without the influence of multiple training passes that could obscure specific I/O patterns. This gave us a deep understanding of how file formats directly impact I/O during training, an important aspect of our research. Furthermore, running multiple epochs could generate a larger volume of data and make the results more complex to interpret concerning I/O patterns. We collected accurate and detailed data essential for our research goals by limiting our experiment to a single epoch. Furthermore, the decision to run only a single epoch to detect the pattern is supported by considering time efficiency and the notion that a single epoch can provide a general representation of I/O patterns observed across multiple epochs.

5.1.4 Analysis of I/O behavior (dataset in HDF5 format)

To analyze the I/O behavior using the dataset in HDF5 format, the same file was shared but with two different access modes. The first was a distributed access, in which each process was responsible for reading a different part of the file. The second mode consisted of all processes reading the entire file. Each of these approaches is explained in more detail below.

Table 4 HDF5 Distributed Sharing: Offset and Request size fragment (Process 0)

Segment	Offset	Length
0	0	8
1	0	16
2	16	80
3	96	512
4	680	512
5	136	544
6	1504	328
7	800	512
8	840	544
9	1106176712	544
10	2536	328
11	854213736	328
⋮	⋮	⋮
143	645271328	512
144	1832	512
145	2416	512
146	1872	544
147	3136	328
148	2864	512
149	1167632408	512
150	1186066600	512
⋮	⋮	⋮
1048	1081595168	512
1049	141348232	6144000
1050	614545544	6144000
1051	159784328	6144000
1052	860358792	6144000
1053	620689544	6144000
⋮	⋮	⋮

HDF5 Distributed sharing:

Figure 11 illustrates the spatial pattern observed in the HDF5 file during its reading by an image recognition application on a node with four processes during training. The processes access the HDF5 file in a distributed manner, reading different portions in parallel, which optimizes input/output (I/O) performance. The read operations, with different offsets and lengths, suggest an access pattern where each process extracts specific fragments of the dataset, with some minor operations focused on internal file navigation (such as metadata, indices, etc.).

Offsets specify the position in the file where data reading begins. The variation in offsets indicates that the processes are accessing different parts of the file, suggesting distributed or segmented access to the dataset.

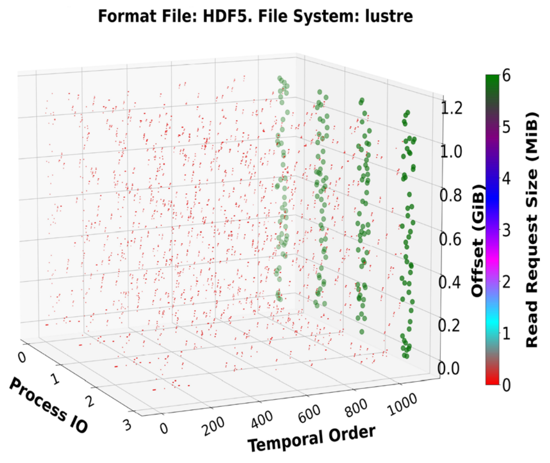


Fig. 12 Temporal Pattern HDF5 Distributed Sharing

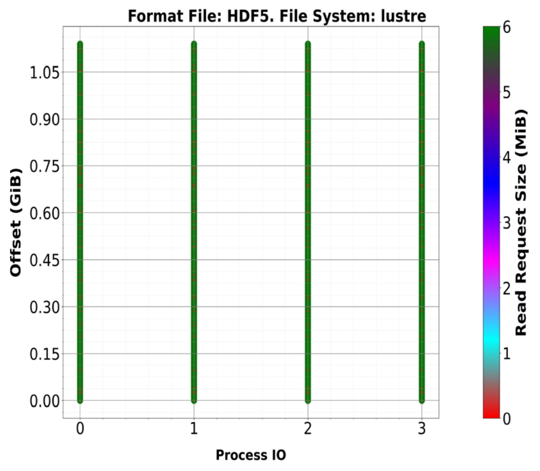


Fig. 13 HDF5 Full sharing: Spatial Pattern

Small reads such as 328, 512, and 544 bytes correspond to accesses to metadata, headers, or small control structures within the HDF5 file. These accesses, represented in red in the figure, may be necessary to navigate or access groups and datasets indices before performing larger data reads.

On the other hand, the 6,144,000 byte (6 MB) reads, represented in green in the figure, are significantly larger and correspond to complete image blocks located within the datasets in the HDF5 file. This is corroborated by comparing the offsets in the trace, which are consistent for each read per process within the datasets. In this context, each process has read 50 datasets, covering a total of 200 directories corresponding to the classes of the Tiny ImageNet-200 dataset in the training directory.

Table 4 shows the behavior of an offset section, where a mix of sequential, strided, and random accesses is observed. Sequential and strided accesses are more common at the beginning, while random accesses increase later, especially when dealing with data at offsets significantly higher than previous accesses.

Table 5 HDF5 Full Sharing: Offset and Request size fragment (Process 0)

Segment	Offset	Length
0	0	8
1	0	16
2	16	80
3	96	512
4	680	512
5	136	544
6	1504	328
7	800	512
8	840	544
9	1106176712	544
10	2536	328
11	854213736	328
⋮	⋮	⋮
143	1130758640	512
144	1130760360	6144000
145	399455560	512
146	417890544	512
147	417890000	544
148	417890936	328
149	417890664	512
150	424035720	6144000
⋮	⋮	⋮
1048	577672840	328
1049	577672568	512
1050	583817352	6144000
1051	497780024	512
1052	503925248	512
1053	503924704	544
⋮	⋮	⋮

Figure 12 illustrates the temporal reading pattern of the HDF5 file. It was observed that accesses with large sizes of 6,144,000 bytes are predominantly found at the end of the order of operations. This is reflected in the figure, where the larger green dots (representing the largest read requests) are clustered towards the end of the time axis. This behavior indicates that larger read operations are performed after a series of smaller accesses. Regarding total values, each of the four processes performed 1099 read operations, resulting in 4396 reads, corresponding to the 1.14 GiB file size.

The repetition of read sizes was consistent. The most common read size was 512 bytes, with 2416 occurrences. The second most frequent read size was 328 bytes, with 948 occurrences, followed by 544, with 816 occurrences. The fourth most common read size was 6,144,000 bytes, with 200 occurrences, the latter being associated with the dataset data. The file was opened four times, once by each process.

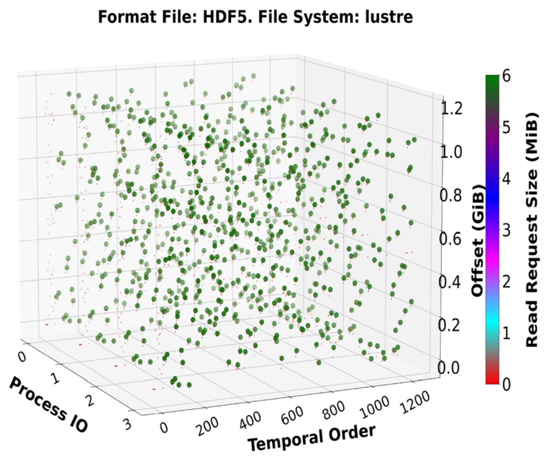


Fig. 14 Temporal Pattern HDF5 Full Sharing

The range of read sizes is as follows: 12 reads were between 0 and 100 bytes, 4,180 reads were between 100 bytes and 1 KiB, four reads were between 1 KiB and 10 KiB, and 200 reads were between 4 MiB and 10 MiB.

HDF5 Full sharing:

Figure 13 illustrates the spatial access pattern observed in the HDF5 file during its reading by a model training application on a node with four processes. In this experiment, the processes accessed the same file concurrently, reading it entirely.

Recurrent sizes of 328, 512, and 544 bytes were observed, corresponding to accesses to metadata, headers, or small control structures within the HDF5 file. A particularly notable size is 6,144,000 bytes (green color), which appears recurrently.

Table 5 shows sequential accesses identified at the beginning, where the offsets increase regularly or with small increments. However, there are also accesses with offsets that show irregular jumps and large sizes, indicating a more random access pattern. These larger accesses are interspersed with smaller ones, reflecting a combination of sequential and random access patterns.

Figure 14 illustrates the temporal read pattern of the HDF5 file. Read requests are more dispersed across than the distributed shared accesses. This suggests that the access patterns are less coordinated and more random in this scenario. The larger requests (indicated by the green dots) are more dispersed along the time axis and not as concentrated at the end, as they were in the previous case. This access pattern is, therefore, less predictable, as we do not see clusters of large read requests at the end of the process, which could generally optimize performance on I/O-intensive systems.

Algorithm 2 Procedure to identify contention

```
1: Open the trace file
2: Extract the accesses by rank:
3:     Identify the accesses made by each rank to the HDF5 file.
4:     Note the offset, the length, and the start and end times (Start(s) and End(s)).
5: Detect overlaps:
6:     Compare the time intervals and offsets to identify overlaps.
7:     If two or more accesses coincide in offset and overlap in time, contention is confirmed.
```

Regarding total values, each of the four processes performed 1249 read operations, resulting in 4996 total reads, corresponding to the 4.57 GiB. Regarding the consistency of reading sizes, the majority were uniform. The most common read size was 512 bytes, occurring 2416 times. The second most frequent read size was 328 bytes, with 948 occurrences, followed by 544, with 816 occurrences. The fourth most common read size was 6,144,000 bytes, with 800 occurrences, the latter being associated with the dataset's data. The file was opened four times, once by each process. The range of read sizes is as follows: 12 reads were between 0 and 100 bytes, 4,180 reads were between 100 bytes and 1 KiB, four reads were between 1 KiB and 10 KiB, and 800 reads were between 4 MiB and 10 MiB.

5.1.5 Comparison of performance based on changing dataset format

The following is a performance comparison of running a deep learning application using different file formats (JPEG and HDF5). Figure 15 illustrates how choosing different file formats and mapping can significantly influence performance results.

The HDF5 distributed sharing method proves to be the most efficient strategy for handling large files in parallel processing scenarios. This efficiency is reflected in the consistently low execution times in all configurations analyzed. The reason behind this superior performance lies in how this method distributes the workload across processes making fewer reads. Each process accesses a specific portion of the file, minimizing contention and avoiding bottlenecks that can arise when multiple processes attempt to access the same resource simultaneously.

In contrast, full HDF5 file sharing, where all processes access the entire file, introduces significant contention. This type of access generates increased competition among processes for input/output resources, resulting in increased execution time. Contention becomes especially problematic as the number of processes and nodes increases, as the underlying storage infrastructure is placed under increased load, unable to handle multiple concurrent requests efficiently.

To observe contention, we analyze the results of the DXT trace obtained with Darshan, following the steps described in the procedure shown in Algorithm 2. In this analysis, we check if there are multiple accesses for each offset. If overlapping accesses are detected in time at the same offset, contention is considered to exist and a message is generated with the corresponding details.

By applying the steps of Algorithm 2, 1003 contentions were detected in HDF5 Full Sharing mode, 600 of which

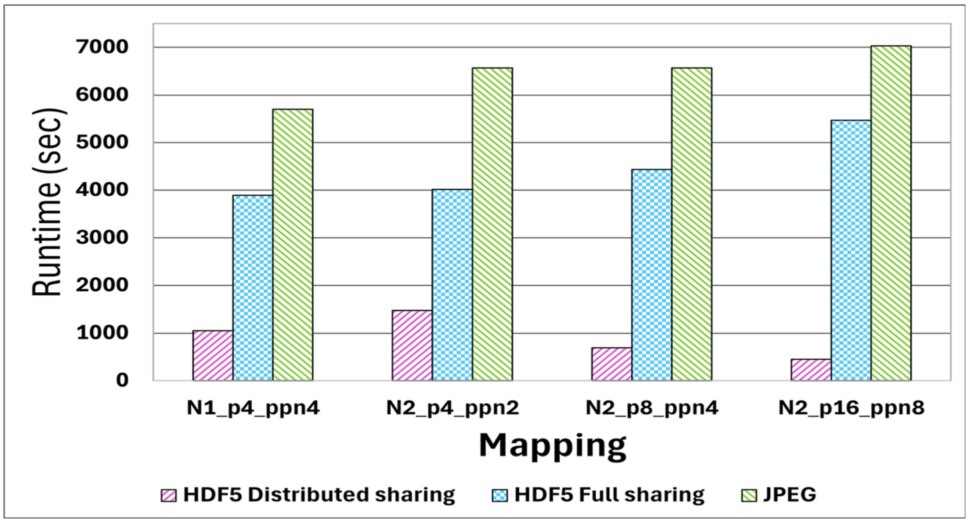
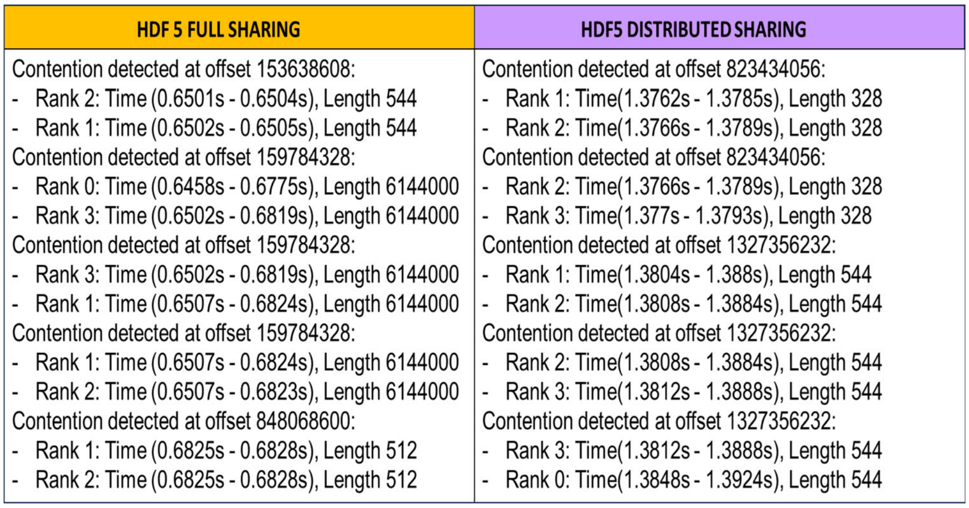


Fig. 15 Comparison of runtime for 5 epochs by storage format and access mode with different mappings on the x-axis, where N is the number of nodes, p is the number of processes, and ppn is the number of processes per node (see table 3)

Fig. 16 Part of the contention detected on the trace



HDF 5 FULL SHARING	HDF5 DISTRIBUTED SHARING
Contention detected at offset 153638608:	Contention detected at offset 823434056:
- Rank 2: Time (0.6501s - 0.6504s), Length 544	- Rank 1: Time(1.3762s - 1.3785s), Length 328
- Rank 1: Time (0.6502s - 0.6505s), Length 544	- Rank 2: Time(1.3766s - 1.3789s), Length 328
Contention detected at offset 159784328:	Contention detected at offset 823434056:
- Rank 0: Time (0.6458s - 0.6775s), Length 6144000	- Rank 2: Time(1.3766s - 1.3789s), Length 328
- Rank 3: Time (0.6502s - 0.6819s), Length 6144000	- Rank 3: Time(1.377s - 1.3793s), Length 328
Contention detected at offset 159784328:	Contention detected at offset 1327356232:
- Rank 3: Time (0.6502s - 0.6819s), Length 6144000	- Rank 1: Time(1.3804s - 1.388s), Length 544
- Rank 1: Time (0.6507s - 0.6824s), Length 6144000	- Rank 2: Time(1.3808s - 1.3884s), Length 544
Contention detected at offset 159784328:	Contention detected at offset 1327356232:
- Rank 1: Time (0.6507s - 0.6824s), Length 6144000	- Rank 2: Time(1.3808s - 1.3884s), Length 544
- Rank 2: Time (0.6507s - 0.6823s), Length 6144000	- Rank 3: Time(1.3812s - 1.3888s), Length 544
Contention detected at offset 848068600:	Contention detected at offset 1327356232:
- Rank 1: Time (0.6825s - 0.6828s), Length 512	- Rank 3: Time(1.3812s - 1.3888s), Length 544
- Rank 2: Time (0.6825s - 0.6828s), Length 512	- Rank 0: Time(1.3848s - 1.3924s), Length 544

had a length of 6,144,000. In the case of HDF5 Distributed Sharing, 491 contentions were identified, all corresponding to small sizes associated with the metadata.

Some of the results obtained are shown in Image 16, where it can be observed that in Full Sharing access mode, the readings of the different processes (ranks) overlapped in time and affected the same offsets or adjacent regions of the file with large sizes of 6,144,000. In the trace, it is evident how several processes read from the same offset at very close times. When two or more processes access the same offset in a short time interval, this may indicate contention.

In contrast, in Distributed Sharing mode, coincident reads at the same offset and similar times were only observed for very small metadata reads. No overlaps in the data were detected, as each process accessed a different offset to perform the read.

These results suggest that overlaps with large read sizes, such as 6,144,000, can lead to higher contention, significantly impacting system performance. This contention may increase if the number of processes and nodes increases, as more processes would try to access the same offset simultaneously, potentially intensifying I/O performance bottlenecks.

On the other hand, the JPEG format, although widely used for storing images, proves less efficient in these parallel processing configurations. The significantly higher execution times in all scenarios suggest that, although the JPEG format is suitable for image storage and transmission in general-purpose systems, it is not optimized for the massive concurrent access common in HPC applications. The JPEG format lacks the internal organization and optimized parallel access capabilities of HDF5, leading to inefficiencies when handled in a distributed processing environment. In addition, JPEG's inherently sequential compression can introduce additional latencies when accessing data in parallel.

5.1.6 Comparison of execution time as a function of storage format and access method in different training epochs

Figure 17 presents the execution time in seconds for three data access methods: HDF5 with distributed sharing, HDF5 with full sharing, and JPEG with independent access, evaluated at three training stages of the model, corresponding to epochs 5, 10 and 15. The HDF5 access method with distributed sharing consistently maintains the shortest execution time at all epochs, positioning it as the most efficient strategy at early and later training stages. This behavior reinforces the idea that distributing the read load across processes significantly reduces contention and optimizes performance.

Although the HDF5 method with full sharing is less efficient than distributed sharing, it still outperforms the JPEG format. This is due to its ability to support some level of parallel access, which decreases contention compared to the sequential or independent access typical of the JPEG format. This design offers a considerable advantage in environments where concurrency in data access is important.

Both methods, HDF5 with full sharing and JPEG show an almost linear increase in execution time as epochs increase. This pattern suggests that these methods may not scale well with increasing epochs, possibly due to the build-up of contention or latency in data access. The observed trend indicates that the HDF5 distributed sharing strategy is the fastest and most scalable among the evaluated options in a prolonged training environment with simultaneous access to large volumes of data.

Analysis of results

Considering the time needed to convert the file format and add it to the training time, the HDF5 method with distributed sharing access is still more efficient than other

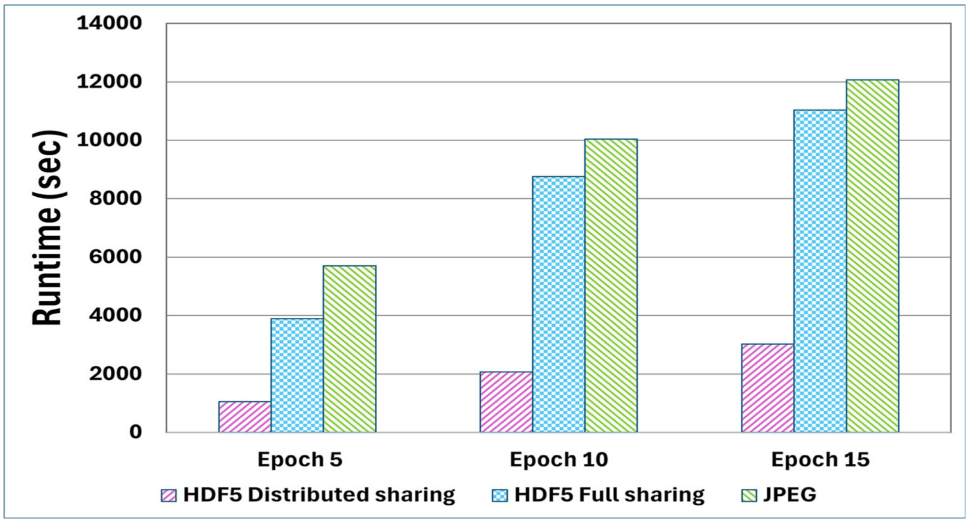


Fig. 17 Comparison of Runtime by Storage Format and Access Mode in Different Epochs (Number of nodes=1, Number of processes=4, Number of processes per node=4)

formats and access modes, as shown in Table 6. The transformation time is only counted once since the same previously transformed data set is reused in subsequent runs.

Table 6 presents the transformation time (38.60 min) for the first run, where training was performed with 5 epochs for the HDF5 file format. It can be observed that the best time (transformation + training) was obtained by HDF5 Distributed Sharing with 56.17 min, compared to HDF5 Full Sharing with 103.48 min and JPEG with 95.04 min, whose time reflects only the training as it is the original format. The total time is highlighted in the table to emphasize thevalues used for comparison between formats. The transformation time is not considered for the remaining runs with 10 and 15 epochs since the same file generated for the first run is used. These results suggest that, in the case studied, converting to the HDF5 distributed format is an advantageous strategy since it provides significant time savings.

It is important to note that this reformatting process is performed only once. Once the dataset has been transformed into the HDF5 format, it can be reused in multiple subsequent trainings, either with different mapping configurations or by increasing the number of epochs, without the need to perform the conversion again. This represents a significant advantage in terms of saving time and resources.

This analysis demonstrates that the format change was worthwhile in this case, as the shared and distributed access method with HDF5 improved data management efficiency and consistently delivered superior performance. These findings underscore the importance of evaluating and, where necessary, optimizing the data storage format to maximize performance in high-performance computing environments.

Table 6 Comparison of times obtained with different formats and access modes

No. Epoch	Times	Times in minutes		
		HDF5 Distributed sharing	HDF5 Full sharing	JPEG
Epoch 5	Training time	17.57	64.88	95.04
	Transformation time	38.60	38.60	—
	Total time	**56.17**	**103.48**	**95.04**
Epoch 10	Training time	34.55	146.03	167.42
Epoch 15	Training time	50.54	184.01	201.32

6 Considerations for the selection and adaptation of Storage Formats in Deep Learning

Based on the previous chapters, where the impact of the storage format on the performance of deep learning applications has been analyzed, it is clear that the right format can be a significant element in optimizing both processing efficiency and the use of computational resources. Therefore, specific criteria should be considered when changing the data format. The following are some of the key considerations that should be taken into account when making a format change to improve performance:

- Data Set Size: When the dataset is considerably large (several GB or TB), selecting a storage format that can efficiently handle large volumes of data is appropriate.
- Number and Size of Files: If the dataset is composed of many small files, switching to a format that allows these files to be grouped into a single container can

significantly improve performance by reducing the number of individual accesses to the file system.
- Data Access Pattern: If the application requires parallel access to the data (multiple processes simultaneously accessing the same data set), a format that efficiently supports this type of access would be more appropriate.
- Metadata and Storage Requirements: If the data includes a large amount of metadata that must be stored and managed along with the main data, a format that natively supports metadata would be a better choice.
- High-Performance Computing (HPC) environments: In HPC systems designed to handle large volumes of data in parallel, switching to formats optimized for these environments is advisable.

7 Conclusions

The choice of file format is an important factor determining the performance and efficiency of deep learning applications on high-performance computing systems. Throughout this study, we have shown that changing file formats can generate I/O behaviors, directly influencing the application's overall performance.

The analyzed case study shows that the proposed strategy, based on transforming the data format to HDF5 with shared and distributed access, effectively improves the data reading efficiency, offering a significant advantage in deep learning on HPC. This transformation optimizes training time and provides long-term benefits by allowing repeated and efficient use of the transformed data sets.

In addition, it is essential to carefully consider when and how to change a data storage format. Factors such as the size of the dataset, the access pattern, and the characteristics of the execution environment must be evaluated to make informed decisions. The pre-processing method we have presented facilitates this decision process by providing a practical tool that transforms the data and optimizes its management and access, ensuring that the format change improves application performance.

Therefore, this study aims to provide a comprehensive view of how file formats, I/O patterns, and file access modes influence the behavior and performance of deep learning applications on HPC systems. It highlights the importance of carefully considered data management to optimize high-performance applications.

Additionally, this study highlights the potential to develop automated tools that dynamically parameterize the format transformation process. These tools could be tailored to the characteristics of both the dataset and the file system, optimizing performance and resource efficiency. Implementing this systematic approach would allow the proposed method to scale to scenarios with greater structural diversity and significantly larger data volumes, thereby enhancing its practical utility.

Our pre-processing strategy represents a practical solution to address these challenges. It highlights the importance of selecting and adapting the appropriate file format in the design and execution of deep learning applications in HPC environments with parallel file systems.

In future work, data redistribution will be explored as an additional preprocessing step to further improve performance. In addition, advanced data loading techniques and the use of aggregators to optimize data loading and processing efficiency on HPC systems will be investigated.

Acknowledgements This research has been supported by the Agencia Estatal de Investigación (AEI), Spain, and the Fondo Europeo de Desarrollo Regional (FEDER) UE, under contract PID2020-112496GB-I00 and partially funded by the Fundacion Escuelas Universitarias Gimbernat (EUG). The author thankfully acknowledges RES resources provided by CESGA in FinisTerrae III to RES-DATA-2022-1-0014.

Author contributions E.P. and B.L. designed and performed the experiments, analyzed the data, designed the methodology, and wrote the manuscript with support from D.R., D.R. and S.M. directed the Project. D.F. and E.L. supervised the Project. All authors discussed the results and contributed to the final manuscript.

Funding Open Access Funding provided by Universitat Autonoma de Barcelona.

Data availability No datasets were generated or analysed during the current study.

Declarations

Competing interests The authors declare no competing interests.

References

1. Behzad, B., Huchette, J., Luu, H., Aydt, R., Koziol, Q., Prabhat, M., Byna, S., Chaarawi, M., Yao, Y.: SC companion: high performance computing. Netw. Storage Anal. (2012). https://doi.org/10.1109/SC.Companion.2012.236

2. Faroukhi, A.Z., El Alaoui, I., Gahi, Y., Amine, A.: Big data monetization throughout big data value chain: a comprehensive review. J. Big Data **7**(1), 1–22 (2020)
3. Chien, S.W., Markidis, S., Sishtla, C.P., Santos, L., Herman, P., Narasimhamurthy, S., Laure, E.: Characterizing deep-learning I/O workloads in TensorFlow. In: 2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS), pp. 54–63. IEEE (2018)
4. Rajesh, N., Bateman, K., Bez, J.L., Byna, S., Kougkas, A., Sun, X.-H.: TunIO: An AI-powered Framework for Optimizing HPC I/O. In: 2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 494–505. IEEE, (2024)
5. Behzad, B., Luu, H.V.T., Huchette, J., Byna, S., Prabhat, Aydt, R., Koziol, Q., Snir, M.: Taming parallel I/O complexity with auto-tuning. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '13. Association for Computing Machinery, New York (2013). https://doi.org/10.1145/2503210.2503278
6. The HDF Group: HDF5 User's Guide. https://docs.hdfgroup.org/hdf5/v1_14/index.html. Accessed 20 March 2024
7. Hudson, G., Léger, A., Niss, B., Sebestyén, I.: JPEG at 25: still going strong. IEEE MultiMed **24**(2), 96–103 (2017). https://doi.org/10.1109/MMUL.2017.38
8. Wallace, G.K. (1992) The JPEG still picture compression standard. IEEE Transactions on Consumer Electronics. https://doi.org/10.1109/30.125072
9. Miano, J.: Compressed Image File Formats JPEG, PNG, GIF, XBM, BMP, 2nd edn. Addison-Wesley, Massachusetts (2000)
10. Chapman, Hall: High performance parallel I/O, 1st edn. CRC Press is an imprint of Taylor & Francis Group an Informa business, Boca Raton (2015)
11. Dragly, S.-A., Hobbi Mobarhan, M., Lepperød, M.E., Tennøe, S., Fyhn, M., Hafting, T., Malthe-Sørenssen, A.: Experimental directory structure (Exdir): an alternative to HDF5 without introducing a new file format. Front. Neuroinf. **12**, 16 (2018)
12. Zheng, H., Vishwanath, V., Koziol, Q., Tang, H., Ravi, J., Mainzer, J., Byna, S.: HDF5 Cache VOL: Efficient and scalable parallel I/O through caching data on node-local storage. In: 2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 61–70 (2022). https://doi.org/10.1109/CCGrid54584.2022.00015
13. Tang, H., Koziol, Q., Ravi, J., Byna, S.: Transparent asynchronous parallel I/O using background threads. IEEE Trans. Parallel Distrib. Syst. **33**(4), 891–902 (2022). https://doi.org/10.1109/TPDS.2021.3090322
14. HPCwire: MLPerf Issues HPC 1.0 Benchmark Results Featuring Impressive Systems (Think Fugaku). (2021). Accessed 26 March 2024 https://www.hpcwire.com/2021/11/19/mlperf-issues-hpc-1-0-benchmark-results-/featuring-impressive-systems-think-fugaku/
15. Bae, M., Jeong, M., Yeo, S., Oh, S., Kwon, O.-K.: I/O performance evaluation of large-scale deep learning on an hpc system. In: 2019 International Conference on High Performance Computing & Simulation (HPCS), pp. 436–439 IEEE (2019)
16. Zhang, Z., Huang, L., Manor, U., Fang, L., Merlo, G., Michoski, C., Cazes, J., Gaffney, N.: FanStore: Enabling efficient and scalable I/O for distributed deep learning. Preprint at https://arxiv.org/abs/quant-ph/1809.1079 (2018)
17. Pumma, S., Si, M., Feng, W.-C., Balaji, P.: Scalable deep learning via i/o analysis and optimization. ACM Trans. Parallel Comput. **6**(2), 34 (2019). https://doi.org/10.1145/3331526
18. Kim, Y., Choi, H., Lee, J., Kim, J.-S., Jei, H., Roh, H.: Efficient large-scale deep learning framework for heterogeneous multi-GPU cluster. In: 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W), pp. 176–181 (2019). https://doi.org/10.1109/FAS-W.2019.00050
19. Byna, S., Chen, Y., Sun, X.-H., Thakur, R., Gropp, W.: Parallel I/O prefetching using MPI file caching and I/O signatures. In: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, pp. 1–12 IEEE (2008)
20. Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A.C., Fei-Fei, L.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. (IJCV) **115**(3), 211–252 (2015). https://doi.org/10.1007/s11263-015-0816-y
21. Parraga, E., Leon, B., Mendez, S., Rexachs, D., Suppi, R., Luque, E.: An empirical method for processing I/O traces to analyze the performance of DL applications. In: Naiouf, M., De Giusti, L., Chichizola, F., Libutti, L. (eds.) Cloud Comput. Big Data and Emerg. Top., pp. 74–90. Springer, Cham (2025)
22. Vijayvargiya, G., Silakari, S., Pandey, R.: A Survey: Various techniques of image compression.https://arxiv.org/abs/quant-ph/1311.6877 (2013)
23. Wang, Z., Lin, Z., Xu, L., Zhao, Y., Xin, J.: Batch images compression algorithm based on the common features. In: 2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI), pp. 1–6 (2017). https://doi.org/10.1109/CISP-BMEI.2017.8301937
24. Carns, P., Harms, K., Allcock, W., Bacon, C., Lang, S., Latham, R., Ross, R.: Understanding and improving computational science storage access through continuous characterization. ACM Trans. Storage **7**(3) (2011). https://doi.org/10.1145/2027066.2027068
25. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Commun. ACM **60**(6), 84–90 (2017)
26. Sergeev, A., Balso, M.D.: Horovod: fast and easy distributed deep learning in TensorFlow. Preprint at https://arxiv.org/abs/quant-ph/1802.05799 (2018)
27. Diederik, P.K.: Adam: A method for stochastic optimization (2014)
28. Le, Y., Yang, X.S.: Tiny imagenet visual recognition challenge. (2015)

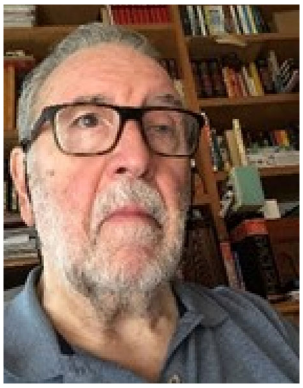
Edixon Parraga Is a research fellow in the Computer Architecture and Operating Systems Department at the Universitat Autònoma de Barcelona (UAB), Spain. His research focuses on input/output (I/O) performance in Artificial Intelligence applications on HPC systems. He studies how access patterns, file formats, and parallel file systems affect the scalability of distributed training. He also works on I/O instrumentation, predictive performance modeling and tuning I/O parameters to optimize application behavior.

Betzabeth Leon , Ph.D., is a professor in the Computer Architecture and Operating Systems Department at the Universitat Autònoma de Barcelona (UAB), Spain. Her research interests include parallel I/O for artificial intelligence applications in High-Performance Computing (HPC) environments. She focuses on performance modeling of applications running on HPC systems, with an emphasis on resource efficiency analysis and fault tolerance mechanisms.

Sandra Mendez is a Research Associate at Barcelona Supercomputing Center in the Performance Tools group. She holds a PhD in High Performance Computing (HPC) by the Universitat Autònoma de Barcelona (UAB), Spain, where she is also an Assistant Lecturer. She is an expert in the evaluation and assessment of monitored patterns, diagnosing bottlenecks in HPC infrastructures and development of strategies for performance optimization of HPC applications. She has also co-authored over 60 peer-reviewed publications.

Dolores Rexachs is a Full Professor of Computer Architecture and Technology at the University Autonoma of Barcelona (UAB), Spain. She is affiliated with the HPC4EAS Research Group. Focuses on computer architecture, with specializations in High-Performance Computing, parallel computer reliability and resilience, parallel Input/Output subsystems, and smart health services. Her research contributes to the UN Sustainable Development Goals.

Daniel Franco is an Associate professor in the Department of Computer Architecture and Operating Systems at the UAB School of Engineering, where he has served as director, vice-director, and head of studies. His research has focused on parallel computing, computer architecture, highperformance interconnection networks, computer simulation, and more, always within the context of competitive national and international research projects.



Emilio Luque is an emeritus professor at the Computer Architecture and Operating System Department at University Autonoma of Barcelona (UAB), Spain. Invited professor in different universities in USA, Asia, Europe and South America, key note speaker in Computer Science Conferences and leader in several research projects founded by the European Union (EU), the Spanish government and different companies. His major research areas are: Simulation and Optimization of Emergency Services in Hospitals (Smart Health Services), Performance and Scalability prediction in HPC systems, Efficient Applications and Fault Tolerance in Parallel Computers. He has been the supervisor of 24 PhD theses, and has published more than 300 papers in technical journals and international conferences.