# Accelerating Deep Learning Training Through Transparent Storage Tiering

Marco Dantas, Diogo Leitão, Peter Cui†, Ricardo Macedo, Xinlian Liu⋆, Weijia Xu†, João Paulo
*INESC TEC & University of Minho* †*University of Texas at Austin* ⋆*Hood College*

***Abstract*—We present MONARCH, a framework-agnostic storage middleware that transparently employs storage tiering to accelerate Deep Learning (DL) training. It leverages existing storage tiers of modern supercomputers (*i.e.,* compute node's local storage and shared parallel file system (PFS)), while considering the I/O patterns of DL frameworks to improve data placement across tiers. MONARCH aims at accelerating DL training and decreasing the I/O pressure imposed over the PFS.**

**We apply MONARCH to TensorFlow and PyTorch, while validating its performance and applicability under different models and dataset sizes. Results show that, even when the training dataset can only be partially stored at local storage, MONARCH reduces TensorFlow's and PyTorch's training time by up to 28% and 37% for I/O-intensive models, respectively. Furthermore, MONARCH decreases the number of I/O operations submitted to the PFS by up to 56%.**

***Index Terms*—I/O optimization, storage tiering, deep learning**

## I. INTRODUCTION

High-performance computing (HPC) infrastructures are increasingly popular to support computational demanding deep learning (DL) training workloads. These workloads are typically backed by large-scale datasets that range from few GiB to several TiB in size and are made of multiple small-sized files. For example, Open Images [1] has around 9 million images and ImageNet-22k [2] has approximately 14 million images.

At the model training phase, data samples are repeatedly read from storage to achieve accurate and unbiased models. However, the long-lived and recurrent access to millions of small files can overload the HPC's shared parallel file system (PFS) (*e.g.,* Lustre [3], BeeGFS [4], GPFS [5]) with the sheer amount of metadata and data requests. This load can also lead to high throughput variability and performance loss for DL jobs and other concurrent jobs accessing the PFS [4], [6]–[9].

To alleviate the small file performance bottleneck, *i.e.,* reduce the number of metadata and data operations submitted to the PFS, DL frameworks (*e.g.,* TensorFlow [10], PyTorch [11], MXNet [12]) support optimized data formats, such as TensorFlow's TFRecords, MXNet's RecordIO, and HDF5 [13], that pack several small-sized files into a single, larger one. Further, to boost the access to training data, these frameworks implement different I/O optimizations, such as in-memory caching, I/O prefetching, and parallel I/O [14]–[17].

Complementary to these optimizations, and since several modern supercomputers include compute nodes equipped with fast local storage mediums (*e.g.,* SSD, NVMe) [18], [19], storage tiering can be used to fully or partially cache datasets locally, reducing the I/O pressure at the PFS and speeding up DL training [20], [21]. However, this paper identifies four challenges that are currently limiting the adoption of this optimization at supercomputers.

**Storage tiering is not available to all DL frameworks.** Most DL frameworks assume training datasets are stored in a single storage backend. In such cases, the decision to move the dataset from the PFS to the local storage mediums must be done manually by users, which in turn, are often not aware of these resources, or even how to use them.

**The full dataset must fit at the faster tier.** Some solutions avoid manual user intervention but require training data to fit entirely at the local storage medium, which is not the case for large datasets [22], [23]. Alternatively, the local disks from several compute nodes can be grouped to provide a caching tier that supports large datasets [24]. However, under single-node DL jobs, this approach requires allocating, and potentially wasting, resources from several compute nodes.

**Intrusiveness for developers and users.** Existing storage tiering solutions addressing the two previous challenges require changing the original codebase of DL frameworks, thus limiting their applicability [21]. These also require understanding and using additional I/O libraries (*e.g.,* custom-made, MPI) for building DL training scripts, limiting user adoption [20].

**DL-specific I/O patterns are unexplored.** Storage tiering systems, such as Hermes [20], are focused towards buffering scientific write workloads at intermediary storage mediums before reaching the PFS. However, DL training workloads are read-oriented, and have specific I/O patterns that should be considered when optimizing data placement over different storage tiers [15], [21]. Namely, the full dataset must be accessed for each training epoch, and each dataset file is read once per epoch. Files may be requested in a randomized order across epochs. Also, when using large file formats (*e.g.*, TFRecords), several I/O read requests are issued to read different data samples packed into a single file.

To address the aforementioned challenges, we propose MONARCH, a framework-agnostic storage tiering middleware for single-node DL training at HPC centers. It enables DL frameworks to transparently leverage local storage mediums of compute nodes, even for datasets that may not fit entirely on such resources. At its core, MONARCH mediates dataset read requests between DL frameworks and HPC storage resources (*i.e.*, local storage and PFS), while providing a data placement strategy that is fine-tuned for the I/O patterns of DL training workloads. Namely, data placement is done as a background task, to avoid adding extra latency at the critical I/O path

of DL frameworks. Further, it prefetches content from large files, stored at the PFS, to faster storage mediums, which not only promotes the use of faster storage resources, but also avoids unnecessary accesses to the PFS. When combined, these contributions *i)* accelerate DL training, *ii)* reduce I/O variability, and *iii)* diminish I/O pressure at the PFS.

By decoupling storage tiering from other optimizations, MONARCH can be combined with other mechanisms currently supported by DL frameworks, such as optimized data formats, I/O caching, prefetching and parallelism. This decoupled design enables porting MONARCH across different DL frameworks, that rely on the POSIX interface to access the training dataset (*e.g.*, TensorFlow, PyTorch), without requiring any changes at their codebase. Finally, it can be used transparently by users without changing how DL training scripts are built.

We implemented a MONARCH prototype and applied it over TensorFlow and PyTorch. The conducted experiments, resorting to different models and dataset sizes, validate that it can indeed solve the aforementioned challenges. Namely, it decreases TensorFlow's and PyTorch's training time by up to 28% and 37%, respectively, for I/O-intensive models and datasets that do not fit entirely at the compute node's local storage. Furthermore, MONARCH is able to reduce I/O variability, and decrease the number of operations submitted to the PFS by up to 56%.

In summary, this paper provides the following contributions:

- An **experimental study** that analyzes and compares the impact of running DL training jobs at the compute node's local storage medium and the PFS (§II).
- **MONARCH**, a transparent and portable storage tiering middleware for accelerating DL training workloads (§III).
- **Implementation** of the MONARCH prototype and its applicability over TensorFlow and PyTorch frameworks (§III). MONARCH is publicly available as an open-source project at **https://github.com/dsrhaslab/monarch**.
- An **experimental evaluation** that showcases the impact of MONARCH in terms of training performance, I/O variability, and operations submitted to the PFS (§IV).

## II. MOTIVATION

For DL models to provide accurate predictions, they must be trained with large and varied datasets. During the training phase, data samples are continuously read from a storage backend, preprocessed in memory, typically by the CPU, and then batched and transferred to the GPU to train the neural network. This phase is divided into training epochs, and in each of these, all dataset samples must be accessed.

HPC users typically store these datasets at the PFS, for different reasons [21], [24]: *i)* users might not be aware of local storage mediums available at compute nodes; *ii)* in many cases, data must be manually copied from the PFS to local storage; *iii)* large datasets may not fit entirely at local storage resources. Therefore, storage tiering should be done automatically and transparently for users to leverage the performance benefits of HPC's local storage resources.

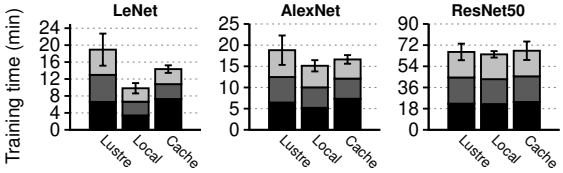



**Fig. 1:** ***Average training time*** *for the* Lustre*,* Local*, and* Cache *setups under LeNet, AlexNet, and ResNet-50 training models. Each column is stacked with the elapsed training time of each training epoch, namely first (* ■ *), second (* ■ *), and third (* ■ *).*

### A. DL training under different storage setups

We conducted an experimental evaluation comparing three different DL training setups currently available to users. Specifically, *i)* dataset samples are served from the PFS (***Lustre***); *ii)* dataset samples are served from the compute node's local storage (***Local***); and *iii)* dataset samples are served initially from the PFS (*i.e.*, during the first training epoch) but are then transparently cached and fully served from the local disk for the remaining epochs (***Cache***).

**Experimental testbed.** Experiments were conducted on a compute node of the Frontera supercomputer [18], which is equipped with two 16-core Intel Xeon processors, four Nvidia Quadro, 128 GiB of RAM, and a single 240 GiB SSD with an accessible 119 GiB partition. Software-wise, it uses CentOS 7.8 with the Linux kernel v3.10 and the `xfs` file system. We memory-limited the compute node to 68 GiB to simulate an environment where the entire dataset would not fit in memory.

The production Lustre file system, available at the Frontera supercomputer, was used as the PFS in these experiments.

**Dataset, models, and DL framework.** We used a truncated version of the ImageNet-1k dataset [25] that includes 900K images (100 GiB), enabling the dataset to fit entirely on the local device. To speedup training performance, we converted the dataset into the TFRecord format, resulting in 1024 TFRecords. To ensure a comprehensive evaluation in terms of workload heterogeneity, experiments included I/O-bound models, namely LeNet [26] and AlexNet [27], and a compute-bound model, namely ResNet-50 [28]. Due to its popularity, we used the TensorFlow DL framework (v.2.3.2) with I/O parallelism, prefetching and parallel preprocessing optimizations enabled. For the *Cache* setup, we also enabled TensorFlow's caching mechanism [23].

**Methodology.** We measured the elapsed training time and resource usage (*i.e.,* average CPU, GPU, memory) of all experiments. These were configured to run for 3 training epochs with a 256 batch size, and simultaneously use all 4 GPUs available in the compute node. Results of each experiment concern the average and standard deviation of 7 runs.

**Training time.** Figure 1 depicts the overall training time, segmented by epochs, under *Lustre*, *Local*, and *Cache* setups for the LeNet, AlexNet, and ResNet-50 models. When compared to *Lustre*, the *Local* setup reduces overall training time for I/O-bound models. Under LeNet, execution time decreases from

18.9 to 9.8 minutes (48%), while for AlexNet, it decreases from 18.9 to 15.1 minutes (20%).

At the *Cache* setup, data samples are cached at the local SSD (*i.e.*, copied from the PFS to the local file system) during the first training epoch. Subsequent epochs fetch data from the local medium, thus reducing the overall training time for the LeNet and AlexNet models by 24% and 12% when compared to *Lustre*. During the first epoch of these two models, training time increases from 6.6 to 7.3 minutes, when compared to *Lustre*. This is due to the extra data copying that must be done between Lustre and the local file system. For the remainder training epochs, the training time is similar to the one observed in the *Local* setup.

For the RestNet-50, all setups perform similarly, ranging from 64 and 67 minutes of execution time, as it imposes less I/O demand [15]. Interestingly, the *Lustre* setup exhibits the highest training time variability across identical runs of each experiment. This is visible for the three models and is due to the fact that the PFS is shared with other jobs executing concurrently at the supercomputer, which can lead to performance variability [4], [6]–[9].

**Resource usage.** For I/O-bound models, CPU and GPU usage are directly related with the speed that data samples are fetched from the corresponding storage backends. Therefore, for both LeNet and AlexNet models, CPU usage increases from 30% (*Lustre*) to 35% (*Cache*). The *Local* setup has the highest CPU usage, namely 57% for LeNet and 43% for AlexNet.

Similarly, for the LeNet model, GPU usage increases from 22% (*Lustre*) to 28% (*Cache*) and to 39% (*Local*). GPU usage for the AlexNet model increases from from 58% (*Lustre*) to 63% (*Cache*) and to 72% (*Local*).

As expected, the compute-bound ResNet model exhibits the same CPU (10%) and GPU (90%) utilization for all the three setups. Further, TensorFlow's memory usage is approximately 10 GiB for all models and setups.

**Summary.** This evaluation shows that serving the training dataset from local storage backends, which are closer to the computation (*i.e., Local* and *Cache*), can *i)* significantly improve DL training performance, particularly under I/O-bound models; *ii)* improve the usage of the compute node's CPU and GPU resources; and *iii)* decrease training performance variability. However, the *Local* setup requires manual intervention from users, while the *Cache* setup provides transparency but it is limited to scenarios where the full training dataset can fit into the available local storage resources.

## III. MONARCH

MONARCH is a framework-agnostic tiering middleware that leverages multiple storage backends at HPC infrastructures. Its design is built under the following core principles.

**Decoupling.** To ensure applicability across different DL frameworks (*e.g.*, TensorFlow, PyTorch) and cross-compatibility with existing I/O optimizations (*e.g.*, data shuffling, prefetching, and parallelism) and storage backends (*e.g.,* local and remote file systems), MONARCH is decoupled from the internal DL framework logic, being implemented as an independent storage middleware.

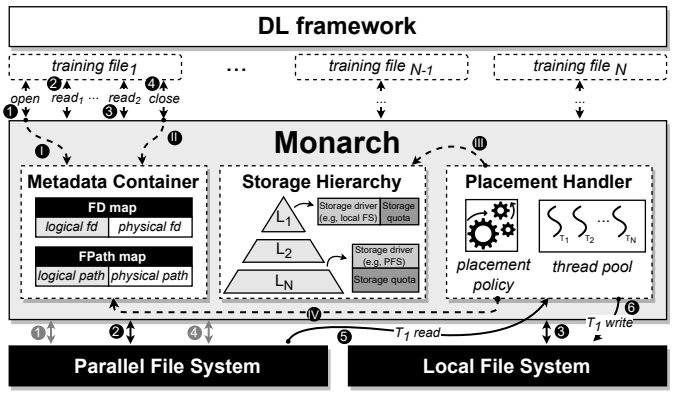



**Fig. 2:** MONARCH*'s architecture and flow of requests.*

**Transparency.** MONARCH does not change how users traditionally build training scripts and use DL frameworks. It can be integrated with existing DL frameworks without requiring any source code changes. This leads to a portable solution that is easy to use at HPC centers.

**Large datasets.** When datasets are large and do not fit completely at local storage mediums, MONARCH automatically chooses the data samples to keep at each storage tier.

**Optimized for DL workloads.** MONARCH is designed to handle I/O patterns specific of DL training. We propose an optimized data placement strategy for workloads that: *i)* read the full dataset for each training epoch; *ii)* may read data samples in random order; and, *iii)* may issue several small-sized I/O requests to read the content of a given training file (*i.e.*, when using large file formats such as TFRecords).

**Training performance and PFS I/O pressure.** MONARCH aims at accelerating the DL training phase, while reducing the I/O operations submitted to the shared PFS. The former is important to improve the quality of service (QoS) of DL users, while the latter is key to improve the QoS of all users resorting to the PFS, as it can be accessed simultaneously by hundreds to thousands of different jobs.

### A. Architecture

As depicted at Fig. 2, MONARCH sits between the DL framework and a hierarchy of storage backends, and follows a POSIX-compliant interface to store and fetch data from both local file systems (mounted on the compute node's local storage) and the PFS (*e.g.,* Lustre).

MONARCH intercepts file read operations submitted by the DL framework and transparently serves the requested content from the most appropriate storage tier. Our solution aims at caching as many training data samples as possible (originally stored at the shared PFS) at the compute node's local device. MONARCH is organized in three main components, namely the *storage hierarchy*, *placement handler*, and *metadata container*.

**Storage hierarchy.** The *storage hierarchy* organizes and manages the storage tiers (or levels) that will be used to read and cache data samples for DL training. Tiers are organized hierarchically, and their order can be configured by users and system administrators. For instance, in the use cases evaluated in this paper (§IV), tiers are organized in descending order in terms of performance, *i.e.*, the local file system is at level 1 and the PFS is at level 2. These, however, could be organized with other criteria, such as storage quota or energy consumption.

Each tier is represented by a *storage driver*, which abstracts the I/O logic performed under a given storage backend. This driver contains a set of properties that allow governing the current state of that backend, including storage path (*i.e.,* file system directory where the training dataset will reside) and available storage quota. This abstraction enables supporting different storage tiers, promoting modularity and extensibility.

The last level (*e.g.,* PFS) holds the full dataset and acts as a read-only data source. Other levels are initialized at the beginning of the training phase without any data samples, being then populated in background by MONARCH.

**Placement handler.** The *placement handler* is responsible for selecting and fetching dataset files to the correct storage tier. We now describe its key features.

*Placement policy.* The selection of the tier where a given file should be placed is addressed with the following policy. Given a *storage hierarchy* of size $N$, the placement starts in descending order, writing dataset files to the first level (*i.e.,* level 1), until reaching its full capacity, moving then to the remainder levels, until all levels are filled ($[1, N-1]$).

In DL workloads, all dataset files are read at each training epoch. Therefore, our placement policy does not perform any file eviction at upper tiers when their storage quota is reached. This decision allows reducing the number of I/O operations served by the PFS tier. Since the dataset access may follow a random distribution (to prevent overfitting [21]), continuously promoting and evicting files (*e.g.,* LRU, FIFO) between storage tiers would increase resource usage and the I/O pressure in the PFS file system, and would not bring performance improvements to the DL training.

*Data fetching and caching.* For each intercepted operation, MONARCH validates if it is destined to a file cached at the upper levels of the storage hierarchy. For non-cached files, the file's content is read from the PFS tier and forwarded to the DL framework. In background, the content of the file is then written to the appropriate upper tier by following the aforementioned placement policy. This asynchronous approach avoids adding latency to the critical I/O path, and allows DL training to start immediately and run simultaneously with our placement algorithm. Further, MONARCH resorts to a dedicated thread pool for this background processing, enabling DL frameworks' I/O requests to be served in parallel.

When the requested file is already cached at a faster storage tier, MONARCH ensures that the requested content is transparently served to the DL framework from such tier. No additional data placement processing occurs for this case.

*Prefetching for large files.* When using large file formats (*e.g.,* TFRecords), the DL framework may submit read operations for obtaining a small portion (*i.e.*, a subset of data samples) of the file's content. In this scenario, MONARCH replies to the DL framework with the requested content, but in background, it prefetches the full file from the PFS to the desired storage level. Thus, when the file is available at the upper storage tier, subsequent read operations to that file can be served from this tier instead. Note that since the file's content is served to the DL framework in the same order as requested, MONARCH does not alter how data is provided to the training workload nor affects the model's accuracy. This aspect, along with the performance benefits of the previous optimizations, are further validated and discussed in §IV.

**Metadata container.** Even though the dataset is *physically* placed over different storage tiers, from the DL framework's perspective (*logical*), it is stored in a single storage backend (*e.g.,* PFS), preventing changes to the DL scripts specified by users or to the framework's codebase. However, to improve training performance and reduce the PFS's I/O pressure, MONARCH aims to always serve requests from faster storage tiers. Therefore, the *metadata container* is responsible for keeping the *logical* and *physical* locations of each dataset file. This information needs to be updated when an existing file is cached at a given storage tier, other than the PFS, and to be consulted when a file is being accessed by the DL framework. Since MONARCH is targeted at frameworks that are using POSIX-compliant backends, we require two different metadata structures. The first maps *logical* file paths to *physical* ones, which is important to redirect system calls, such as `open` and `close`. The second maps *logical* file descriptors to *physical* ones to redirect system calls, such as `pread` and `mmap`. Each metadata entry sizes at 100B. While this entails additional memory consumption, our experiments (§IV) demonstrate that its impact is minimal and justified under I/O-bound workloads.

### B. Operation Flow

We now describe MONARCH's operation flow (Fig. 2).

**Initialization.** Before execution, the system designer specifies the storage tiers that should be considered in a configuration file. For example, MONARCH can be configured with two storage tiers — level 1 respects to the compute node's local file system that is backed by a local SSD drive, while level 2 points to the dataset location at the shared PFS. When the training phase starts, a MONARCH instance is initialized. To initialize the *metadata container*, MONARCH traverses the directory where the dataset resides (level 2) and collects the location (*file path*) and size (*file size*) of each file.

**I/O calls interception and handling.** During the training phase, MONARCH intercepts POSIX calls from the DL framework, including `open`, `pread`, `mmap`, and `close`. Upon an `open` (❶), MONARCH verifies the *metadata container* for the path where that file is stored. If the file is persisted at level 2, which is always the case for files being accessed for the first time, then the request is forwarded to the corresponding

*file path* at the PFS. The resulting file descriptor (*fd*) is then stored at the *metadata container* (❶) and forwarded to the DL framework. If the file is cached at level 1, which can be consulted at the *metadata container*, the request is sent to the *file path* at the local file system, and the resulting *fd* is equally stored at the *metadata container* (❶). Note that the *fd* returned to the DL framework is always the *logical* (original) one associated with the PFS, which is available at the *metadata container*. Again, this decision makes the process of data placement completely transparent to the DL framework.

After opening a file, the DL framework will submit one or more requests (*e.g.*, `pread`, `mmap`) to access the content of that file (❷ and ❸). These are intercepted by MONARCH and redirected to the corresponding storage tier. The mapping between *logical* and *physical fds* is available at the *metadata container*. The content read by MONARCH is then forwarded to the DL framework.

Upon a `close` (❹), MONARCH redirects it to the appropriate storage tier, and forwards the reply back to the framework. Moreover, the metadata entry mapping that *logical* and *physical fd* is deleted at the *metadata container* (⓫).

**Background data fetching and placement.** The data placement is triggered when the content of a given file, which is not yet available at level 1, is requested (read) by the DL framework from level 2. If there is enough free storage quota at level 1 (ⅢⅠ), the requested file's content is then written asynchronously to that level by a background thread. When a small portion of a large file is being requested by the DL framework, MONARCH' background thread will prefetch the full content of the file from level 2 to level 1 (❺ and (❻)).

When the full content for the requested file is available at level 1, the *metadata container* is updated regarding the new *physical file path* for that file (Ⅳ), while the storage quota for that tier is updated (ⅢⅠ). Moreover, if the file is currently being accessed by the DL framework (*i.e.*, an *open* call was submitted and the corresponding *fd* has not been closed yet), the file now persisted at level 1 is opened by MONARCH, and the *logical fd* to *physical fd* metadata mapping is updated accordingly. This enables subsequent read operations from the DL framework to that file to be served by level 1 (such as ❸), instead of level 2, thus further reducing the number of I/O calls redirected to the PFS. This optimization is applicable for scenarios where the DL framework submits multiple `read` requests to a large file for fetching different data samples.

Updates at the *metadata container* structures are thread-safe as both background (*i.e.*, data placement) and foreground (*i.e.*, DL framework requests) operations access these concurrently.

### C. Implementation and Applicability

We have implemented a MONARCH prototype with 3K lines of C++14 code. We used the C++ Thread Pool Library (CTPL) (version 0.0.2) [29] for implementing the thread pool of the *placement handler* module. The *metadata container* lookup tables use both the Abseil (v20210324.2) [30] and Intel Threading Building Blocks Concurrent HashMap (v2021.2.0) [31] libraries. These structures provide a thread-safe environment for concurrent operations, and are kept in memory due to performance considerations. Such design does not compromise the fault tolerance of our solution because, if a DL job fails, the *metadata container* information can be recovered from the data persisted at the PFS.

MONARCH uses `LD_PRELOAD` to transparently intercept POSIX calls (destined towards the *logical path*) and route them to the *physical data path*. Specifically, we replaced the `open`, `pread`, `mmap`, and `close` POSIX calls, supported by `glibc`, by ones that are serviced by MONARCH. We found that supporting this set of calls is sufficient to attend the requirements of the workloads presented in (§IV). MONARCH is publicly available as an open-source project at **https://github.com/dsrhaslab/monarch**.

## IV. EVALUATION

Our evaluation seeks to answer the following questions:

- Is MONARCH applicable over different DL frameworks?
- Can MONARCH improve training performance of different DL models and dataset sizes?
- Can MONARCH reduce the I/O pressure on the PFS?
- Does MONARCH impact DL training accuracy?

**Experimental testbed.** The experimental setup, models, and methodology used in these experiments are the same as those described in §II. Two different datasets based on the ImageNet-1k were used: a *small* version, sizing at 100 GiB (§II), and a *large* version, sizing at 200 GiB. The *large* version was used to assess a scenario where the dataset cannot fit entirely in the compute node's local storage and memory. The dataset was converted into TFRecords, resulting in 2048 training files.

**MONARCH configuration.** MONARCH was configured with 6 threads for the *placement handler*'s thread pool and two storage levels for the *storage hierarchy*. Level 1 corresponds to the compute node's `xfs` file system, mounted on top of a local SSD partition with 115 GiB. Level 2 corresponds to the directory where the dataset is stored at Lustre.

**DL frameworks configurations.** To demonstrate the applicability of MONARCH, it was evaluated under TensorFlow (v2.3.2) and PyTorch (v1.6.0). TensorFlow was set with the same configurations as in §II. PyTorch experiments were conducted in conjunction with the DALI framework (v.1.5.0) [14]. DALI was configured with 16 threads for the LeNet and AlexNet models, and 8 threads for the ResNet model (more threads would cause internal memory allocation errors).

For the remainder of this section, we analyze MONARCH regarding training performance, volume of data and metadata operations submitted to the PFS (*i.e.,* Lustre), and resource usage for each combination of DL framework (TensorFlow and PyTorch) and dataset size (100 GiB and 200 GiB).

### A. TensorFlow 100 GiB

**Training performance.** Fig. 3 depicts the training performance for all experimental scenarios. When compared to

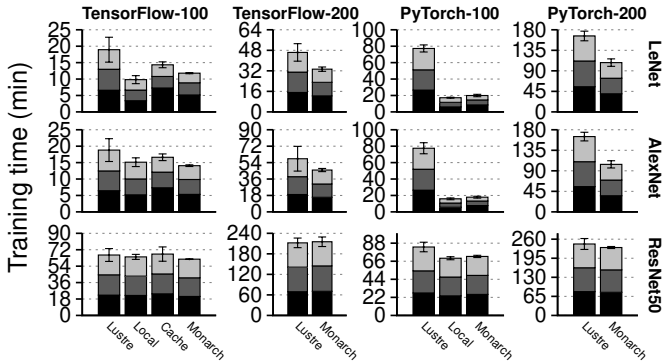



**Fig. 3:** ***Average training time*** *of Lustre, Local, Cache, and* MONARCH *setups in TensorFlow and PyTorch, under different training models (LeNet, AlexNet, ResNet-50) and dataset sizes (100 GiB and 200 GiB). Each column is stacked with the elapsed time of each training epoch, namely first (* ■ *), second (* ■ *), and third (* ■ *).*

*Lustre*, MONARCH significantly improves the overall training performance for I/O-bound models, decreasing training time by 38% (7.1 min) for LeNet and 26% (4.9 min) for AlexNet. For ResNet-50, all setups perform similarly.

For the first training epoch (steps $[0, 3500]$), under I/O-bound models, MONARCH achieves better performance than *Lustre* and *Cache*. This is due to MONARCH's *large file prefetching* mechanism (§III-A, *placement handler*). Specifically, when a `read` call is submitted to a given TFRecord, MONARCH fetches the whole file from the PFS. Under this scenario, both reads (from the DL framework) and writes (submitted to the local storage tier by MONARCH) are buffered at the compute node's page cache. As depicted in Fig. 4a, this optimization has particular impact in the first half of the first epoch, where MONARCH experiences significantly higher throughput than the aforementioned setups. However, for the second half, as the page cache fills, MONARCH's throughput degrades while matching the performance of *Lustre* and *Cache* setups. Because the ingestion rate of the DL framework is higher than the flushing rate of dirty pages to local storage, reads start being submitted to the PFS, as the requested files are not yet available at local storage (as described in §III-A). Under the LeNet model, this behavior is also manifested at the beginning of the second training epoch, since there is accumulated backlog (*i.e.,* dirty pages) from the first epoch still being written to the local disk and competing with read requests being done over the same storage medium.

For the second $(]3500, 7000])$ and third $(]7000, 10500])$ training epochs, when the full dataset is persisted at the local tier, MONARCH experiences similar performance as of the *Cache* and *Local* setups. When compared to *Lustre*, MONARCH reduces training time by up to 46% (5.7 min) and 29% (3.6 min) for LeNet and AlexNet, respectively.

**PFS operations.** As depicted in Fig. 5a, due to the *storage tiering* and *large file prefetching* mechanisms, MONARCH significantly reduces the number of `read` calls directed to the PFS. The *Lustre* setup submits approximately $360,000$ `read` calls per epoch, while *Cache* only submits these during the first epoch, since the dataset will then be served from the local storage tier.

In MONARCH, under I/O-bound models, the number of operations submitted to the PFS can be analyzed in three phases, similarly to the training performance. At a first phase, due to the *large file prefetching* mechanism, `read` calls are large, fetching the whole file from the PFS, and forthcoming reads are mainly served from the compute node's page cache. Then, when the page cache fills, MONARCH submits small-sized `read` calls to PFS, while simultaneously storing the dataset in local storage. Finally, when the full dataset is available from the local tier, no more `read` calls are submitted to the PFS. These PFS operations are related to the network traffic generated with the DL training job. For the ResNet-50 model, since it is compute-bound, MONARCH's *prefetching mechanism* is able to fetch all data samples timely, only submitting as many `read` calls as the number of existing TFRecords (1024).

A decrease is also noticeable for metadata operations, namely `open` and `close`. As depicted in Fig. 5b, for MONARCH and *Cache* setups, all operations are concentrated in the first training epoch, performing a single `open` and `close` call for each training file. *Lustre*, on the other hand, repeats this behavior at each training epoch. Additionally, to obtain the necessary information to populate the *metadata container* (*e.g.,* file size), MONARCH performs 1024 additional `getattr` operations (*i.e.,* one `getattr` call per file).

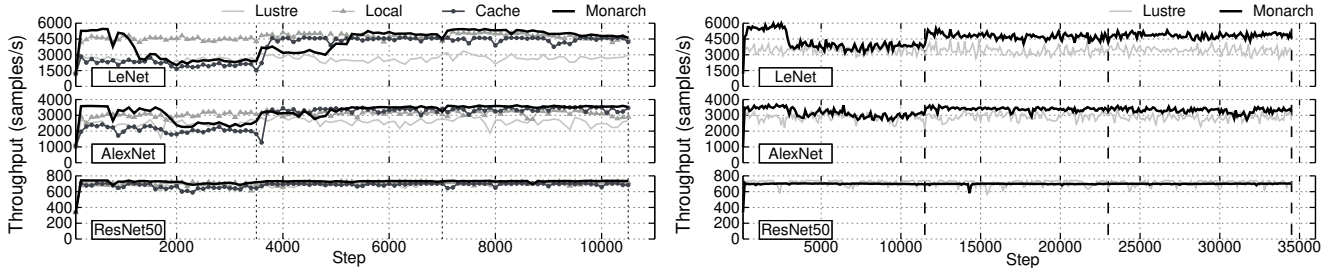
**Resource usage.** Because MONARCH can service training samples faster to the DL framework, it demonstrates the second highest CPU and GPU utilization (being surpassed by *Local*). Specifically, it achieves a CPU and GPU usage of approximately 45% and 33% for LeNet model, 40% and 73% for the AlexNet, and 10% and 92% for ResNet. Regarding memory consumption, MONARCH performs identically has remainder setups (*i.e.,* 10 GiB).

### *B. TensorFlow 200 GiB*

For the 200 GiB dataset, only MONARCH and *Lustre* setups were considered, since both *Cache* and *Local* require the full dataset to fit in the local storage tier.

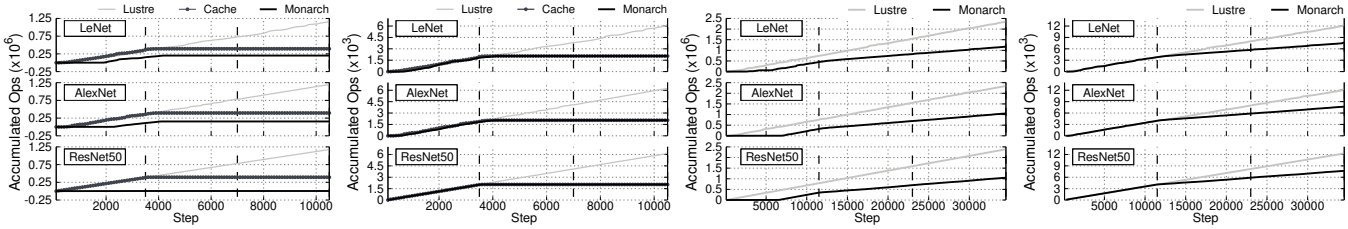
**Training performance.** As depicted in Fig. 3, MONARCH improves training performance under I/O-bound models, decreasing training time by 28% (13 min) for LeNet and 21% (12.5 min) for AlexNet, when compared to *Lustre*. During the first training epoch ($[0, 11500]$), as depicted in Fig. 4b, MONARCH experiences a throughput degradation due to the page cache filling up and the local storage tier achieving its quota. For the remainder epochs, MONARCH serves the DL framework `read` calls from both local and remote storage tiers. For the ResNet-50 model, MONARCH and *Lustre* perform similarly.

**PFS operations.** As depicted in Fig. 5c, the *Lustre* setup submits approximately 2.4 million `read` requests to the PFS across all training epochs. MONARCH is able to significantly reduce this value since a large portion of the dataset is stored in the local storage tier (115 GiB). Specifically, MONARCH




**(a)** *TensorFlow 100 GiB.* **(b)** *TensorFlow 200 GiB.*

**Fig. 4:** ***TensorFlow training performance.*** *Throughput, in samples per second, of Lustre, Local, Cache, and* MONARCH *setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB (a.) and 200 GiB (b.) sized datasets.*




**(a)** *Read calls, 100GiB.* **(b)** *Metadata calls, 100GiB.* **(c)** *Read calls, 200GiB.* **(d)** *Metadata calls, 200GiB.*

**Fig. 5:** ***TensorFlow PFS operations.*** *Accumulated read (a. and c.) and metadata (b. and d.) operations submitted to the PFS for the Lustre, Cache, and* MONARCH *setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB (a. and b.) and 200 GiB (c. and d.) sized datasets.*

reduces PFS read operations by 50% for LeNet and 56% for both AlexNet and ResNet-50. After the first training epoch, contrary to the 100 GiB dataset experiments, MONARCH continues submitting operations to the PFS to access the data samples that could not fit in the local storage tier. Metadata operations manifest the same behavior, as depicted in Fig. 5d.

**Resource usage.** MONARCH is able to increase CPU and GPU efficiency when compared to *Lustre*. In more detail, CPU usage increases from 36% and 31% (*Lustre*) to 48% and 37% (MONARCH) for LeNet and AlexNet, respectively. GPU usage increases from 30% and 63% (*Lustre*) to 40% and 76% (MONARCH). For ResNet-50, both setups exhibit similar CPU (9%) and GPU (90%) usage. Regarding memory consumption, both setups perform similarly (*i.e.,* 10 GiB).

### *C. PyTorch 100 GiB*

Unlike TensorFlow, PyTorch does not include a persistent caching optimization. Thus, experiments were conducted over *Lustre*, *Local*, and MONARCH setups.

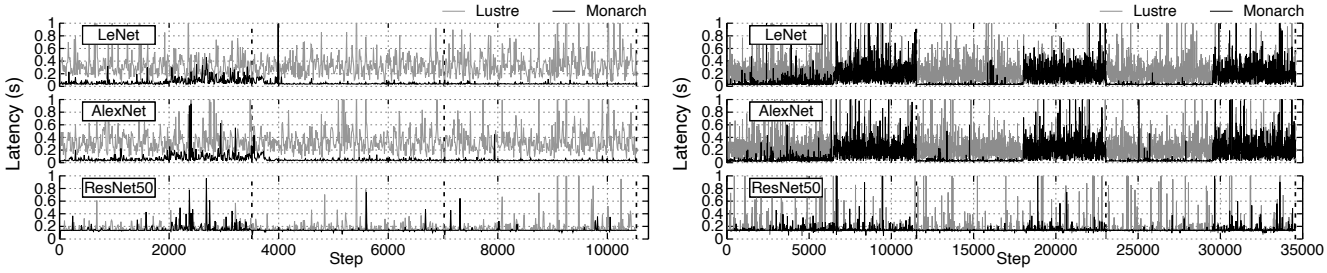
**Training performance.** As depicted in Fig. 3, PyTorch exhibits higher training times than TensorFlow for all three models, specially for the *Lustre* setup and I/O-bound models.

The *Local* setup trains LeNet, AlexNet and ResNet-50 in 17.5, 16.1 and 69.9 minutes, respectively. MONARCH, when compared to *Local*, exhibits similar training execution times. When compared to *Lustre*, MONARCH significantly improves overall training performance for I/O-bound models, decreasing training time by 74% (57.3 min) for LeNet and 77% (59.5 min) for AlexNet. For ResNet-50, MONARCH reduces training time from 84 to 72.4 minutes (14% reduction). Interestingly, when using PyTorch, the ResNet-50 model also becomes I/O-bound, thus explaining the performance improvement of MONARCH.

*Local* achieves sustained latency throughout the overall execution and across all training models, never exceeding 0.2 seconds. To ease illustration, *Local* results were not included in Fig. 6a. For the first training epoch (steps $[0, 3510]$), when compared to *Lustre*, MONARCH is able to reduce training time by 68% (18 min), 71% (18.7 min), and 6% (1.8 min) for LeNet, AlexNet, and ResNet-50, respectively. Similarly to the results observed in §IV-A, this is due to MONARCH's *large file prefetching* mechanism. Again, during the second half of the first epoch, the compute node's page cache fills (with dirty pages) and `read` calls start being submitted to the PFS, as the requested files are not yet available at the local storage tier, leading MONARCH to experience latency spikes.

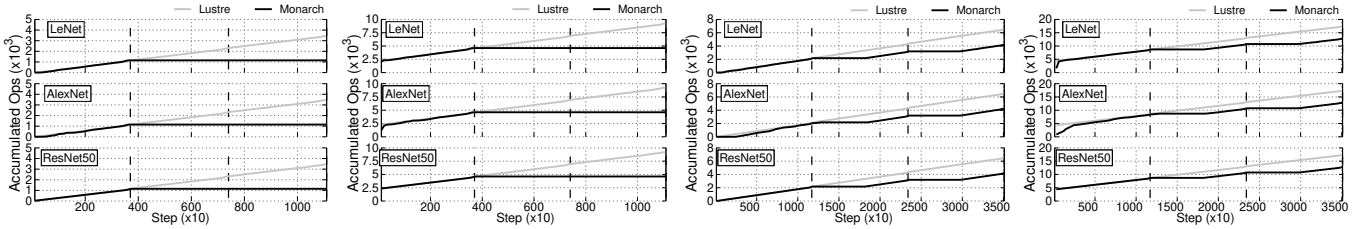
For the second $(]3510, 7020])$ and third $(]7020, 10530])$ epochs, since the full dataset is available at the local tier, MONARCH ensures sustained latency, improving training times by 77% (39.3 min), 80% (40.9 min) and 17% (9.7 min) for LeNet, AlexNet and ResNet-50, respectively.

**PFS operations.** PyTorch uses the `mmap` system call to map whole training files to memory. Contrary to TensorFlow, which performs multiple explicit `read` calls per file, PyTorch's I/O is performed implicitly when it attempts to access the in-memory data (as a result of `mmap`) and a page fault occurs, resulting in the data samples being copied on demand. As depicted in Figs. 7a and 7b, MONARCH's prefetching mechanism significantly reduces the calls directed to the PFS. In detail, *Lustre* submits a total of 3,461 `mmap` and 9,234 (`open` and `close`) metadata calls, while MONARCH only submits 1,152 and 4,618, respectively. Since the `mmap` system call performs




**(a)** *PyTorch 100GiB.* **(b)** *PyTorch 200GiB.*

**Fig. 6:** ***PyTorch training performance.*** *Latency, in seconds, of Lustre and* MONARCH *setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB (a.) and 200 GiB (b.) sized datasets.*




**(a)** *Mmap calls, 100GiB.* **(b)** *Metadata calls, 100GiB.* **(c)** *Mmap calls, 200GiB.* **(d)** *Metadata calls, 200GiB.*

**Fig. 7:** ***PyTorch PFS operations.*** *Accumulated mmap (a. and c.) and metadata (b. and d.) operations submitted to the PFS for Lustre and* MONARCH *setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB (a. and b.) and 200 GiB (c. and d.) sized datasets.*

implicit I/O we have not traced the amount of bytes read. Moreover, MONARCH operations to the PFS are all done in the first training epoch, since after that, all requests are served from the local storage tier.

**Resource usage.** MONARCH shows the second highest CPU and GPU usage, as expected. For the LeNet and AlexNet models, CPU ranges from 5% (*Lustre*), to 22% (MONARCH) and to 26% (*Local*) while, for ResNet-50, it increases from 7% (*Lustre*) to 9% (MONARCH) and to 10% (*Local*). For LeNet, GPU utilization ranges from 14% (*Lustre*), to 50% (MONARCH), and to 58% (*Local*) while, for AlexNet, it increases from 9% (*Lustre*), to 35% (MONARCH), and to 41% (*Local*). For ResNet-50, it ranges from 75% (*Lustre*), to 85% (MONARCH), and to 88% (*Local*). All setups exhibit similar memory consumption, using 10 GiB for LeNet and AlexNet, and 8 GiB for ResNet-50.

### *D. PyTorch 200 GiB*

As in IV-B, for the 200 GiB dataset, only *Lustre* and MONARCH setups were considered.

**Training time.** As depicted in Fig. 3, MONARCH significantly improves training performance under I/O-bound models, decreasing training time by 35% (58 min) and 37% (61.2 min) under LeNet and AlexNet, respectively. For ResNet-50, training time is reduced by 5% (12 min).

Looking at the performance over time (Fig. 6b), MONARCH's latency degrades at the second half of each training epoch. The reason behind this is twofold. First, MONARCH caches approximately half of the training dataset (56%, which corresponds to the storage quota at the compute node's disk) in the local storage tier, serving DL requests from local resources rather than the PFS. Second, when combining PyTorch and DALI, TFRecords are read from storage sequentially and the shuffling process is made in-memory. This leads to a deterministic storage I/O pattern across training epochs (*i.e.,* the first half of the dataset is served from the local storage tier, while the remainder is read from the PFS). Given this sequential access pattern, to further optimize training performance, MONARCH could evict and prefetch samples based on the deterministic order that these are requested. However, as discussed in §III-A, this would increase the I/O pressure at the PFS, since with an eviction policy, MONARCH would always submit operations to the PFS regardless of the training epoch.

**PFS operations.** Similarly to IV-B, the number of operations submitted by MONARCH to the PFS is directly related with the portion of the dataset stored at the local storage tier (*i.e.,* 56%). As depicted in Fig. 7c, during the first training epoch, both *Lustre* and MONARCH perform 1,890 `mmap` calls. However, for the remainder epochs, while *Lustre*'s `mmap` calls increase linearly, MONARCH submits 2,322, representing a 50% reduction. For the second and third epochs (Fig. 7d), MONARCH reduces the number of combined `open` and `close` metadata calls from 9,274 (*Lustre*) to 4,640.

**Resource usage.** MONARCH increases CPU and GPU efficiency when compared with *Lustre*. For LeNet and AlexNet, CPU increases from 6% (*Lustre*) to 10% (MONARCH). For ResNet-50, CPU usage is 7% for both setups.

For LeNet, GPU utilization goes from 20% (*Lustre*) to 31% (MONARCH). For AlexNet, it increases from 13% (*Lustre*) to 21% (MONARCH). For ResNet-50, it ranges from 83%

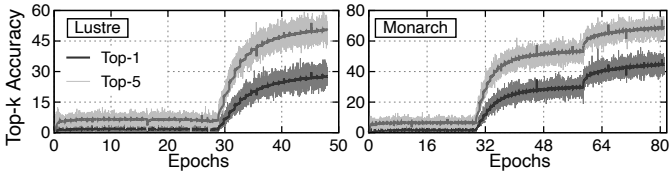



**Fig. 8:** ***Top-1 and top-5 accuracy*** *results for PyTorch with Lustre and* MONARCH *setups the AlexNet model and 200 GiB dataset, over a 48 hours period.*

(*Lustre*) to 87% (MONARCH). Memory consumption results and conclusions are identical to those presented in §IV-C.

### *E. Long run and accuracy analysis*

MONARCH's training performance and accuracy were assessed for a 48 hours long training workload (time limit for regular user jobs at the Frontera supercomputer). The AlexNet model was chosen, along with the PyTorch deployment and the 200 GiB ImageNet-1k dataset used in §IV-D. Next, we compare the results for the *Lustre* and MONARCH setups.

As depicted in Fig. 8, in 48 hours, *Lustre* completes 48 training epochs and reaches *Top-1* and *Top-5* accuracies of 37% and 61%, respectively. MONARCH, on the other hand, completes the same set of epochs in 28 hours (a reduction of 20 hours), while achieving similar *Top-1* and *Top-5* accuracies, namely 38% and 63%.

For the full workload (*i.e.,* 48 hours), MONARCH completes 81 epochs and achieves *Top-1* and *Top-5* accuracies of 51% and 75%. This shows that, for the same time frame, MONARCH can increase the number of epochs and, consequently, the accuracy of trained models.

While this experiment is based on the PyTorch deployment and AlexNet model, for other combinations of frameworks and models (*e.g.*, TensorFlow, LeNet), one would also experience performance improvements proportional to the results discussed in previous sections.

### *F. Summary*

Besides showcasing MONARCH's applicability to different frameworks, models and dataset sizes, the previous experiments validate three key aspects: *i)* MONARCH is able to reduce TensorFlow's and PyTorch's training time by up to 28% and 37% for I/O-bound models; *ii)* for both compute and I/O-intensive models, MONARCH reduces the number of data and metadata operations submitted to the PFS by up to 56%. This is key to ensure stable storage performance for DL workloads and other jobs using the PFS; and, *iii)* MONARCH does not impact the accuracy of DL workloads; in fact, it enables running more training epochs, and consequently achieving better accuracy values, for limited time frames.

## V. RELATED WORK

The DL storage bottleneck is currently a relevant and open research issue that has inspired different I/O optimizations.

**Data loading and preprocessing.** Some proposals improve DL data loading and preprocessing efficiency by resorting to different caching and prefetching algorithms. DALI [14] supports direct I/O prefetching from storage to GPUs. Pumma et al. [16] optimize Caffe's LMDB I/O subsystem to improve the mapping and caching of training data from storage to memory. CoorDL [17] provides insights on storage I/O data stalls and mitigates them by providing a new in-memory caching policy. PRISMA [15] proposes a Software-Defined Storage data plane that performs data prefetching to memory.

While our solution leverages ideas from these works (*e.g.,* caching, applicability to different frameworks), it is focused on using the available storage resources at the supercomputer to accelerate DL training performance and reduce the I/O pressure on the shared PFS. Therefore, these are orthogonal to our work and can even be used in conjunction with it.

**Data substitution and staging.** Other solutions employ data substitution techniques [32]–[34] where training samples being served to DL frameworks are replaced by others (*e.g.,* cached samples) that are faster to access. These techniques are useful for scenarios where several jobs are training models from the same dataset (*i.e.,* shared dataset). Differently, MONARCH optimizations are designed for single-node training scenarios where, to improve accuracy, each file of the dataset must be read once per epoch. In this scenario, if the cache size is relatively small when compared to the full dataset, data substitution techniques either require accessing the PFS multiple times or may lead to fetching the same files repeatedly at each training epoch, thus potentially impacting training accuracy.

Serizawa and Tatebe [22] use the local disks of compute nodes to fully cache datasets to be trained with the Chainer framework. Fanstore [24] aggregates the local storage of several compute nodes to enable data sharing in distributed training environments. Finally, Diesel [35] resorts to local storage mediums and an external distributed key-value store to cache data and metadata information. MONARCH is designed for single-node training and provides a simpler solution that avoids allocating additional resources and orchestrating complex data and metadata staging areas, while not assuming that the dataset fits entirely on faster storage tiers.

**Storage tiering.** NoPFS [21] uses a performance model to proactively fetch training samples, to different storage tiers, before these are requested by the DL framework. However, NoPFS is intrusive to both developers and users, as it requires changing the original source code of DL frameworks and the way training scripts are specified. Contrarily, MONARCH focuses on providing storage tiering, outsourcing the proactive data fetching to built-in mechanisms already present in DL frameworks, or provided by external solutions such as DALI. This enables MONARCH to be non-intrusive for DL frameworks, and avoids changing the way users build DL scripts.

Hermes [20] provides a storage tiering solution for buffering scientific write workloads at intermediary storage mediums before reaching the PFS. In contrast, MONARCH is targeted towards read-oriented DL training workloads. As discussed in the paper, these have specific I/O patterns (*e.g.*, full dataset is read for each training epoch, random I/O accesses) that change the way data samples must be placed across storage tiers.

## VI. Conclusion

This paper presents MONARCH, a storage tiering middleware for accelerating DL training and reducing the I/O pressure and variability imposed in the shared PFS. To achieve this, and promote a wider adoption of storage tiering at HPC infrastructures, MONARCH builds upon four main principles: *i)* it leverages faster local storage mediums, available at compute nodes, to fully or partially cache the training data samples; *ii)* it does so automatically and without changing the way users build their DL training scripts; *iii)* it is portable across different frameworks without requiring source code modifications; and *iv)* it provides data placement mechanisms that are optimized for the I/O patterns present at DL training workloads.

To validate the applicability and performance of MONARCH, we developed a prototype and applied it over TensorFlow and PyTorch frameworks. Results show that TensorFlow's and PyTorch's training time can be reduced by up to 28% and 37% for I/O-intensive models, even for large datasets, that can only be partially cached at local storage mediums. Further, MONARCH is able to reduce the number of I/O operations submitted to the PFS by up to 56%.

## Acknowledgments

We thank the anonymous reviewers for their insightful comments and feedback. We thank TACC for providing access to computational resources of Frontera. We thank Cláudia Brito and Cláudia Correia for their valuable input in initial versions of this work. This work was supported by the Portuguese Foundation for Science and Technology and the European Regional Development Fund, through the PhD Fellowship SFRH/BD/146059/2019 and projects BigHPC (POCI-01-0247-FEDER-045924) and PAStor (UTA-EXPL/CA/0075/2019).

## References

[1] "Open Images Dataset," 2017. [Online]. Available: https://github.com/cvdfoundation/open-images-dataset

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[3] P. Schwan, "Lustre: Building a File System for 1000-node Clusters," in *Proceedings of the Linux Symposium*, 2003.

[4] 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 *International Conference on Parallel Processing*, 2019.

[5] F. B. Schmuck and R. L. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *USENIX Conference on File and Storage Technologies*, 2002.

[6] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing Variability in the IO Performance of Petascale Storage Systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[7] J. Han, L. Xu, M. M. Rafique, A. R. Butt, and S.-H. Lim, "A Quantitative Study of Deep Learning Training on Heterogeneous Supercomputers," in *IEEE International Conference on Cluster Computing*, 2019.

[8] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *International Parallel and Distributed Processing Symposium*, 2016.

[9] G. K. Lockwood, S. Snyder, T. Wang, S. Byna, P. Carns, and N. J. Wright, "A Year in the Life of a Parallel File System," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[10] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[11] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic Differentiation in PyTorch," 2017.

[12] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems," 2015.

[13] T. H. Group, "Hierarchical data format version 5," 2000-2021. [Online]. Available: https://www.hdfgroup.org/solutions/hdf5/

[14] "NVIDIA DALI." [Online]. Available: https://developer.nvidia.com/dali

[15] R. Macedo, C. Correia, M. Dantas, C. Brito, W. Xu, Y. Tanimura, J. Haga, and J. Paulo, "The Case for Storage Optimization Decoupling in Deep Learning Frameworks," in *IEEE International Conference on Cluster Computing*, 2021.

[16] S. Pumma, M. Si, W.-C. Feng, and P. Balaji, "Scalable Deep Learning via I/O Analysis and Optimization," *ACM Transactions on Parallel Computing*, vol. 1, no. 1, 2019.

[17] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram, "Analyzing and Mitigating Data Stalls in DNN Training," *Proceedings of the VLDB Endowment*, vol. 14, no. 5, 2021.

[18] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The Evolution of Leadership Computing at the National Science Foundation," in *Practice and Experience in Advanced Research Computing*, 2020.

[19] "AI Bridging Cloud Infrastructure." [Online]. Available: https://abci.ai

[20] A. Kougkas, H. Devarajan, and X.-H. Sun, "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System," in *Symposium on High-Performance Parallel and Distributed Computing*, 2018.

[21] N. Dryden, R. Böhringer, T. Ben-Nun, and T. Hoefler, "Clairvoyant Prefetching for Distributed Machine Learning I/O," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2021.

[22] K. Serizawa and O. Tatebe, "Accelerating Machine Learning I/O by Overlapping Data Staging and Mini-Batch Generations," in *IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, 2019.

[23] "TensorFlow API: tf.data.Dataset.cache." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/data/Dataset#cache

[24] Z. Zhang, L. Huang, U. Manor, L. Fang, G. Merlo, C. Michoski, J. Cazes, and N. Gaffney, "FanStore: Enabling Efficient and Scalable I/O for Distributed Deep Learning," 2018.

[25] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, "Imagenet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, 2015.

[26] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner *et al.*, "Gradient-based Learning Applied to Document Recognition," *Proceedings of the IEEE*, vol. 86, no. 11, 1998.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012.

[28] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[29] "CTPL." [Online]. Available: https://github.com/vit-vit/CTPL

[30] "Abseil." [Online]. Available: https://abseil.io/

[31] "OneTBB." [Online]. Available: https://github.com/oneapi-src/oneTBB

[32] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl, "Faster neural network training with data echoing," 2020.

[33] Y. Zhu, F. Chowdhury, H. Fu, A. Moody, K. Mohror, K. Sato, and W. Yu, "Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems," in *IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2018.

[34] A. V. Kumar and M. Sivathanu, "Quiver: An Informed Storage Cache for Deep Learning," in *USENIX Conference on File and Storage Technologies*, 2020.

[35] L. Wang, S. Ye, B. Yang, Y. Lu, H. Zhang, S. Yan, and Q. Luo, "DIESEL: A Dataset-Based Distributed Storage and Caching System for Large-Scale Deep Learning Training," in *International Conference on Parallel Processing*, 2020.