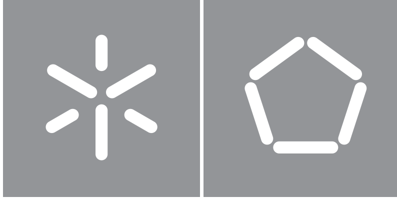
Universidade do Minho
Escola de Engenharia

Marco Filipe Leitão Dantas

Accelerating Deep Learning Training on High-Performance Computing with Storage Tiering

March, 2022



Universidade do Minho
Escola de Engenharia

Marco Filipe Leitão Dantas

Accelerating Deep Learning Training on High-Performance Computing with Storage Tiering

Master's Dissertation
Integrated Master in Informatics Engineering

Work supervised by
João Tiago Medeiros Paulo
Rui Carlos Mendes de Oliveira

March, 2022

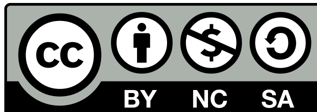
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositóriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Atribuição-NãoComercial-CompartilhaIgual 4.0 Internacional
CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.pt

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Acknowledgements

After all these years my journey is complete and it was only possible with the help of a lot of people that I cannot forget to express my gratitude.

First of all, I would like to thank Professor Rui Oliveira for supporting this dissertation. Professor João Paulo, for the opportunity to work on a project that allowed me to learn so much, for the constant guidance, knowledge and encouragement that kept me focused on properly delivering this work. Ricardo Macedo, who constantly contributed with great advice and expertise, having a major impact in the project. I cannot thank enough the huge help of both João and Ricardo. You guys had so much patience with me. I was always filled with doubts and failures and you helped me overcome them with our numerous meetings.

I want to thank Diogo Leitão, for taking the time to contribute with engineering skills to the project, as well as Cláudia Correia that gave us many necessary know-hows to have a clearer and easier start to this work.

To the Texas Advanced Computing Center (TACC) team, especially Weijia Xu, Xinlian Liu and Peter Cui, who gave relevant input to the course of the project and provided us the necessary infrastructure to conduct this work's experiments.

A heartfelt thank you to my parents who fully supported and contributed to my education. They were always by my side during the difficult times and were restless to make my aspirations possible.

To Tiago and Gabriel, who shared a roof with me through all this years. You gave me countless hours of happiness and enjoyment, as well as support. Thank you so much.

With this journey's end I need to thank all of the friends that I made in the Academy. It was a pleasure spending time and work with them across all of my curricular units. They truly made the past years an unforgettable learning experience.

Finally, I thank the Portuguese Foundation for Science and Technology (FCT) for funding this dissertation within the PAStor project (UTA-EXPL/CA/0075/2019).

Abstract

Accelerating Deep Learning Training on High-Performance Computing with Storage Tiering

Deep Learning (DL) has become fundamental to the advancement of several areas, such as computer vision, natural language processing and expert systems. Utilizing DL techniques demands vast amounts of data and processing power, which raises challenges to the training performance of DL models. *High-Performance Computing (HPC)* systems are becoming increasingly popular to support DL training, by offering extensive computing capabilities, however, due to convenience and usability, many DL jobs running on these infrastructures resort to the shared *Parallel File System (PFS)* for storing and accessing training data. Under such scenario, where multiple Input/Output (I/O)-intensive applications operate concurrently, the PFS can quickly get saturated with simultaneous storage requests and become a critical performance bottleneck, leading to throughput variability and performance loss.

To solve these issues, this dissertation presents a storage middleware agnostic to any DL solution, Monarch, that deploys storage tiering to accelerate DL models' training performance and decrease the I/O pressure imposed over the PFS. It leverages from existing storage tiers of supercomputers (*e.g.,* compute node's local storage, shared PFS), as well as the I/O patterns of DL solutions to improve data placement across storage tiers. Furthermore, this middleware is non-intrusive and easily installed in HPC centers, thus enabling its wide adoption and applicability.

The performance and applicability of Monarch are validated with the TensorFlow and PyTorch DL frameworks. Results show that, when the training dataset can only be partially stored at the local storage tier, Monarch decreases TensorFlow's and PyTorch's training time by up to 28% and 37% for I/O-intensive models, respectively. Furthermore, Monarch can reduce the number of I/O operations submitted to the PFS by up to 56%.

Keywords: I/O optimization, Storage Tiering, Deep Learning.

Resumo

Aceleração do Treino de Aprendizagem Profunda em Computação Avançada com Armazenamento por Camadas

Aprendizagem Profunda (AP) tornou-se fundamental para o avanço de diversas áreas, como visão por computadores, processamento de linguagem natural e sistemas especializados. A utilização de técnicas de AP requer vastas quantidades de dados e de poder de processamento, o que impõe desafios ao desempenho do treino de modelos de AP. Os sistemas de *Computação de Alto Desempenho (CAD)* estão a tornar-se cada vez mais populares para suportar treino de AP, uma vez que oferecem extensos recursos de computação, contudo, por razões de conveniência e usabilidade, muitas tarefas de AP que correm nestas infraestruturas recorrem a *Sistema de Ficheiros Paralelos (SFP)* para armazenar e aceder a dados de treino. Neste cenário, onde múltiplas aplicações intensivas em Entrada/Saída (E/S) operam concorrentemente, o SFP pode ficar saturado com os pedidos de armazenamento simultâneos e tornar-se um gargalo de desempenho crítico, levando à variabilidade do *débito* e perda de performance.

Para resolver estes problemas, esta dissertação propõe um *middleware* de armazenamento agnóstico a qualquer solução de AP, Monarch, que implementa armazenamento por camadas, para acelerar o desempenho do treino de AP e diminuir a pressão de E/S imposta sobre o SFP. Este sistema aproveita camadas de armazenamento existentes em supercomputadores (*p.ex.,* armazenamento local do nó de computação, SFP partilhado), assim como o padrão de E/S das soluções de AP para melhorar a colocação dos dados ao longo das camadas de armazenamento. Para além disso, este middleware é não-intrusivo e facilmente instalado em centros de CAD, permitindo, deste modo, a sua ampla adoção e aplicabilidade.

O desempenho e aplicabilidade do Monarch são validados recorrendo às soluções de AP TensorFlow e PyTorch. Os resultados mostram que, quando o conjunto de dados de treino apenas pode ser parcialmente armazenado na camada de armazenamento local, o Monarch diminui o tempo de treino com TensorFlow e PyTorch entre 28% e 37%, para modelos intensivos em E/S, respetivamente. Para além disso, o Monarch consegue reduzir o número de operações de E/S submetidas para o SFP até 56%

Palavras-chave: Otimização de E/S, Armazenamento por Camadas, Aprendizagem Profunda.

Contents

List of Figures ix

List of Tables xi

Listings xii

Acronyms xiii

1 INTRODUCTION 1

1.1 Problem . . . 2

1.2 Objectives . . . 4

1.3 Contributions . . . 5

1.4 Results . . . 6

1.5 Document structure . . . 7

2 STATE OF THE ART 8

2.1 Background . . . 8

2.1.1 Machine Learning . . . 8

2.1.2 Deep Learning . . . 9

2.1.3 Overfitting . . . 11

2.1.4 Model's evaluation metrics . . . 12

2.1.5 Deep Learning Frameworks . . . 13

2.1.6 Deep Learning Execution Time overview . . . 13

2.1.7 Deep Learning on HPC systems . . . 15

2.1.8 Storage Solutions for Deep Learning . . . 18

2.1.9 TensorFlow's Data Loading Solutions overview . . . 20

2.1.10 PyTorch's Data Loading Solutions Overview . . . 22

2.2 Related work . . . 23

2.2.1 Data Ingestion Pipeline . . . 23

2.2.2 Parallel I/O . . . 24

2.2.3 I/O Buffering 24
2.3 Summary 32

3 PRELIMINARY EXPERIMENTS 35
3.1 Experimental Setup 36
3.2 Results 37
3.3 Summary 39

4 MONARCH 41
4.1 Architecture 42
4.1.1 Storage hierarchy 42
4.1.2 Placement handler 43
4.1.3 Metadata container 44
4.2 Operation Flow 45
4.3 Implementation 47
4.3.1 Configuration 47
4.3.2 Applicability Across DL Frameworks 48
4.3.3 Threading and Background Processing 48
4.3.4 Metadata Management 49
4.4 Summary 49

5 EXPERIMENTAL EVALUATION 51
5.1 Experimental Setup 51
5.2 Results 53
5.2.1 TensorFlow 100 GiB 53
5.2.2 TensorFlow 200 GiB 56
5.2.3 PyTorch 100 GiB 59
5.2.4 PyTorch 200 GiB 62
5.2.5 Long run and accuracy analysis 64
5.2.6 Discussion 65

6 CONCLUSION 67
6.1 Prospects for Future Work 68

Bibliography 70

List of Figures

1 Deep Learning layers representation. 10
2 Deep Learning simplified training process steps. 10
3 Example of the dataset access of a training loop that uses Mini-batch Gradient Descent (Mini-batch GD) and *global shuffling*. 11
4 Overfitting, appropriate fitting and underfitting for a classification problem. 12
5 Deep Learning generic data-flow and execution time components. 14
6 Deep Learning generic pipelined data-flow 15
7 Parallel File System being used by multiple nodes. 16
8 Simplified example of cache *thrashing* originated by the LRU replacement policy existent in the Page Cache, considering two epochs, cache quota of 2 units, and a worst case scenario for a random access pattern. 17
9 Random access to an optimized file format. 19
10 Sequential access to an optimized file format. 20
11 Interleaved access to an optimized file format. 21
12 TensorFlow's file system adapters. 22

13 Average training time for the *Lustre*, *Local*, and *Cache* setups under LeNet, AlexNet, and ResNet-50 training models. 38
14 Central Process Unit (CPU) utilization of each model in the different setups. 39
15 Graphical Processing Unit (GPU) utilization of each model in the different setups. 39

16 Monarch's cache eviction policy compared with LRU. 44
17 Monarch's architecture and flow of requests. 45

18 Average training time for the *Lustre*, *Local*, and *Cache* setups under LeNet, AlexNet, and ResNet-50 training models for the TensorFlow 100 GiB scenario. 53
19 TensorFlow's throughput, in samples *per second*, of *Lustre*, *Local*, *Cache*, and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB. 54
20 TensorFlow 100 GiB PFS operations. 55

21 TensorFlow 100 GiB resource utilization. 56
22 Average training time for the *Lustre* and *Monarch* setups under LeNet, AlexNet, and ResNet-50 training models for the TensorFlow 200 GiB scenario. 57
23 TensorFlow's throughput, in samples *per second*, of *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 200 GiB. 57
24 TensorFlow 200 GiB PFS operations. 58
25 TensorFlow 200 GiB resource utilization. 59
26 Average training time for the *Lustre*, *Local*, and *Cache* setups under LeNet, AlexNet, and ResNet-50 training models for the PyTorch 100 GiB scenario. 59
27 PyTorch's latency, in seconds, of *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB. 60
28 PyTorch 100 GiB PFS operations. 61
29 PyTorch 100 GiB resource utilization. 62
30 Average training time for the *Lustre*, *Local*, and *Cache* setups under LeNet, AlexNet, and ResNet-50 training models for the PyTorch 200 GiB scenario. 62
31 PyTorch's latency, in seconds, of *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 200 GiB. 63
32 PyTorch 200 GiB PFS operations. 63
33 PyTorch 200 GiB resource utilization. 64
34 Top-1 and top-5 *accuracy*. 65

List of Tables

1 Specifications of the experimental environment. 36

Listings

1 Setting up the storage tiers hierarchy with Monarch. 47

2 Examples of Monarch supported POSIX calls. 48

Acronyms

AI Artificial Intelligence

ANN Artificial Neural Network

AP Aprendizagem Profunda

API Application Programming Interface

CAD Computação de Alto Desempenho

CNN Concolutional Neural Networks

CPU Central Process Unit

DFS Distributed File System

DI Department of Computer Science and Informatics

DL Deep Learning

DLFS Deep Learning File System

DLT Deep Learning Training

DNN Deep Neural Network

DTT Data Tracking Tool

E/S Entrada/Saída

EOO Entropy-aware Opportunistic Ordering

FIFO First In, First Out

GPU Graphical Processing Unit

HDD Hard Disk Drive

HDF5 Hierarchical Data Format version 5

HFDS Hadoop Distributed File System

HPC High-Performance Computing

HSM Hierarchical Storage Management

HTTP Hypertext Transfer Protocol

I/O Input/Output

ILSVRC2012 ImageNet Large Scale Visual Recognition Challenge 2012

KV Key-Value

LDL Locality-aware Data Loading

LFU Least Frequently Used

LMDB Lightning Memory-Mapped Database

LRU Least Recently Used

Mini-batch GD Mini-batch Gradient Descent

ML Machine Learning

MP Single Node Multi-Process

MPI Message Passing Interface

NVM Non-Volatile Memory

NVMe Non-Volatile Memory Express

OS Operating System

PFS Parallel File System

PLF Prefetching for Large Files

QoS Quality of Service

RAM Random-Access Memory

RDMA Remote Direct Memory Access

ReLU Rectified Linear Unit

REST Representational State Transfer

RNN Recurrent Neural Networks

RTT Round Trip Time

SDS Software-Defined Storage

SFP Sistema de Ficheiros Paralelos

SGD Stochastic Gradient Descent

SP Single Node Single Process

SPDK Storage Performance Development Kit

SQL Structured Query Language

SSD Solid-State Drive

TACC Texas Advanced Computing Center

TanH Hyperbolic Tangent

TCP Transmission Control Protocol

TLU Threshold Logic Unit

TPU Tensor Processing Unit

VM Virtual Machine

Chapter 1

INTRODUCTION

Artificial Intelligence (AI) has been gathering interest from the scientific community and industry [99], leading to technological solutions and innovations. For example, computer vision is becoming a major investment for companies [99] to develop face detection and recognition solutions, used in so many state-of-the-art smartphones. Another example is human pose estimation, that can be applied to create augmented reality in the fashion industry [11] or to identify behaviors in the surveillance industry [2]. Moreover, natural language processing is also achieving rapid progress by having AI systems with high language capabilities [99]. As an example, Google and Microsoft have both deployed the BERT [22] language model in their search engines. Another example is the use of AI in biology, where DeepMind's AlphaFold applied DL to solve the protein fold problem [40].

It is challenging to specify a universal characterization of the AI field, since it branches out to many specific subdomains with alternative approaches for problem-solving. Some of them are attached to the evolutionary computation field, which mainly uses meta-heuristics or stochastic optimizations to achieve high-quality solutions to optimization problems, for example, *genetic algorithms* [7]. Others can be associated with automatic improvement of the designed system through accumulated experience, such as Machine Learning (ML).

ML uses *Linear Algebra* for a systematic representation of the knowledge that a computer can understand, and *Multivariate Calculus*, to achieve a mathematical optimization of a given function. Thus, ML allows the construction of AI for multiple real-world applications and uses real-world data, specific to a problem, to do so. In this case, the computer, through an ML model will learn how to produce an output, based on a set of inputs (*i.e,* training dataset), acquiring learning capabilities [32].

ML is composed by a considerable variety of models. The ones based on Deep Learning (DL) are well know to deal with text, sound and image problems [99]. DL is based on Artificial Neural Networks (ANNs),

which are slightly inspired by biology, more specifically by the brain architecture, although DL models are not models of the brain. The term "Deep" merely states that the ANN, in this case, contains multiple connected *hidden layers* in its topology. Each layer is a set of neurons or nodes that hold specific state (*weights*) in their connections. During a training phase the neurons state will be optimized to achieve good results at solving the predefined problem (*e.g.,* distinguish an image of a dog from an image of a cat). As in ML, this phase is only possible when a problem-specific dataset is used. This dataset will be repeatedly read, as a whole, to provide the training data inputs.

Complex DL models must be trained with large datasets in order to be accurate [27, 82]. The exponential growth in data, supported by the rise of the internet, as a way to collect and distribute data, was without a doubt one of the pivotal reasons to allow the DL field to bloom [27]. These datasets are often comprised of hundreds of GiB, and even several TiB in size, for example, ImageNet-22k [21] (1.5 TiB), Open Images [65] (18 TiB) and YouTube-8M [4] (1.6 TiB). Each data sample is usually read in a randomized order to ensure model convergence and optimal *accuracy* values [57, 59].

A realistic DL workload is formed by the combination of high computation and data ingestion times, leading to prolonged training phases. Many solutions were developed and used by companies, as well as the scientific community, to solve this performance issue, which further helped the rise of the DL field. For example, and on the hardware side, accelerators like NVIDIA Graphical Processing Units (GPUs) [89] and Google Tensor Processing Units (TPUs) [39] can improve the performance of training DL models. Likewise, on the software side, DL frameworks, such as Keras[41], TensorFlow[1] and PyTorch[66], are widely adopted to facilitate the development of DL training scripts. These solutions provide efficient algorithms to train DL models, as well as better approaches to take advantage of the available computational resources. An example of the latter is the availability of distributed training techniques, that use multiple GPUs and/or multiple nodes for the training process. These frameworks can also integrate in their source code other external libraries, for example cuDNN [14], which is used to further improve the DL training performance when using GPUs.

1.1 Problem

It is important, due to the storage and computational requirements of DL training (*i.e.,* caused by the large amounts of data and complex models, respectively), to use infrastructures with specialized hardware to run DL jobs. HPC infrastructures are a great example to tackle the performance issues of DL training, by accommodating thousands of compute nodes, where DL jobs can be deployed to effectively train DL models. These compute nodes have high-speed interconnect networks for communications, and can possess multiple GPUs. Furthermore, compute nodes have easy access to one or multiple PFS backends (*e.g.,* Lustre [77], BeeGFS [17], GPFS [76]), which are shared by all HPC users to store and access their information.

Indeed, HPC users that run DL jobs at compute nodes, generally use the available Parallel File System (PFS) backend to store their large training datasets, despite contrary recommendations [28]. These storage infrastructures are highly scalable, but when stressed by many users their performance can deteriorate, leading to performance degradation [17, 53, 54, 96]. Moreover, DL datasets, in addition to being large, are often comprised of millions of data samples. For example, Open Images has around 9 million images and ImageNet-22k has approximately 14 million images. As such, PFSs, like other storage solutions, are affected by the *small files problem* [51, 75], and the long-lived and recurrent access to millions of small files (*i.e.,* both data and metadata requests) during the training process of a DL model can cause a massive impact in the supercomputer's PFS, as stated in [100], a Texas Advanced Computing Center (TACC) research project:

"As the dataset grows larger, the metadata and data traffic of thousands of directories and millions of files can easily saturate the existing shared file system due to the high access frequency, concurrency, and the sustained Input/Output (I/O) behavior."

In fact, DL jobs storage access can be underwhelming in these infrastructures, causing performance losses for the training phase, which evidences the existence of an I/O bottleneck for these jobs. Furthermore, the presence of multiple DL workloads can lead to performance degradation and high throughput variability on concurrent jobs that also depend on the PFS performance. To alleviate the *small files* I/O bottleneck,(*i.e.,* reduce the number of metadata and data operations submitted to the PFS), optimized data formats are used. These formats pack several small-sized files into a single, larger one. This type of data representation diminishes the I/O bottleneck by allowing the use of large and sequential reads, leading to the reduction of the number of files being accessed and, consequently, the number of metadata operations required during the training phase. Some DL frameworks support optimized data formats, such as TensorFlow's TFRecords [86] or MXNet's RecordIO [13, 72]. On the other hand, third-party libraries can also be used, such as HDF5 [30].

Further, to boost the access to training data, DL frameworks also implement different I/O optimizations, such as in-memory caching, I/O prefetching, and parallel I/O. These optimizations are often conveyed through convenient programming interfaces, such as TensorFlow's tf.data[85] and PyTorch's [66] DataLoader [70], or by using third-party libraries, such as DALI [61].

Complementary to these optimizations, and since several modern supercomputers include compute nodes equipped with fast local storage mediums (*e.g.,* Solid-State Drive (SSD), Non-Volatile Memory Express (NVMe)) [5, 80], 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 [24, 44]. However, this work identifies four main challenges that are currently limiting the adoption of storage tiering 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, moving the dataset from the PFS to the local storage mediums must be done by users, either by manually copying the dataset or by providing the DL framework, if possible, custom logic to read data samples. However, users are often not aware of these local storage resources, or do not know how to use them correctly.

The full dataset must fit at the faster tier. While some approaches avoid manual intervention from users, the training data must fit into the compute node's local disk, which is not the case for large DL datasets [78, 84]. In some approaches, the local disks from several compute nodes can be grouped to provide a caching tier that supports large datasets [100]. However, under single-node DL jobs, such solutions require allocating, and potentially wasting, resources from several compute nodes.

Intrusiveness for developers and users. Solutions addressing the previous challenges can require changing the original codebase of DL frameworks, thus limiting their applicability. Also, these solutions require understanding and using additional I/O libraries (*e.g.,* custom-made, Message Passing Interface (MPI)) for building DL training scripts, limiting user adoption.

DL-specific I/O patterns are unexplored. Current storage tiering approaches 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. 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 optimized data 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, a framework-agnostic storage tiering middleware is necessary. This solution should enable DL frameworks, in single-node training scenarios, to transparently leverage local storage mediums of compute nodes, even for datasets that may not fit entirely on such resources.

1.2 Objectives

Taking into consideration the problems described in the previous section, this dissertation has three main objectives. First, it aims at accelerating single-node DL training. This performance improvement is targeted at training phases that access supercomputer's PFS storage infrastructure, thus being affected by the previously described I/O bottleneck. This will allow DL developers at supercomputers to train DL models in

shorter periods of times and further increase the DL training *evaluation metric* result (*e.g., accuracy*). The latter applies when regular user's jobs have a time limit to run at the supercomputer (*e.g.,* Frontera [80]).

Second, considering the perspective of the HPC cluster, it is important to establish the additional goal of mitigating the number of requests that are issued to the PFS (*i.e.,* both data and metadata). This objective is necessary to reduce the performance impact of DL jobs on the HPC infrastructure's shared PFS, thus improving the Quality of Service (QoS) of all users that resort to the PFS. Further, by depending less on accessing the PFS, DL jobs, will be less affected by the I/O variability of that storage infrastructure, leading to a more stable performance of DL training across different jobs.

Finally, it is important to ensure transparency for DL users and DL frameworks cross-applicability. With this goal, this dissertation's proposed system can be applied over different DL frameworks with distinct I/O optimizations. Moreover, DL training scripts will not have the need to be modified and DL users do not have the necessity to describe a complex set of configurations to use the proposed system. All of this leads to the system's wider adoption.

1.3 Contributions

To accomplish the objectives stated above the following contributions are made in this work:

- The first contribution is an **experimental study** that analyzes and compares the impact of running DL training jobs at the compute node's local storage medium and at a supercomputer's PFS. This experiment will serve as a foundation and baseline to determine the real advantages of using the local storage medium as a faster storage tier.

- The second contribution is **Monarch**, a framework-agnostic storage tiering middleware for single-node DL training at HPC centers. Monarch 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 jobs, that are performing model training under datasets that hold different types of data, such as raw small files or optimized data formats (*i.e.,* TFRecords, RecordIO). 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. This decision promotes the use of faster storage resources while avoiding unnecessary data accesses at the PFS.

  When combined, this middleware's mechanisms *i)* accelerate DL training time, *ii)* reduce I/O variability, and *iii)* diminish I/O pressure at the PFS. Moreover, by decoupling storage tiering from other

I/O 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 to their codebase.

Finally, it can be utilized transparently by users without requiring any changes to the way they build their DL training scripts or requiring any complex system configuration.

- **Implementation** of Monarch, which relies on the *LD_PRELOAD* technique to be portable across DL frameworks without the need to change their source code or DL developers training scripts. Monarch prototype is ready to be used with the TensorFlow and PyTorch (with DALI [61] enabled) frameworks.

- An **experimental evaluation** that showcases the impact of Monarch in terms of training performance, I/O variability, and number of operations submitted to the PFS. This evaluation was conducted on a realistic scenario, using the Frontera [80] supercomputer and with the TensorFlow and PyTorch frameworks to train the LeNet [50], AlexNet [45] and ResNet [33] models. Monarch allows decreasing TensorFlow's and PyTorch's training times by up to 28% and 37%, respectively, for I/O-bound 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 shared PFS by up to 56%.

1.4 Results

Has a result of this work the two following scientific papers were published:

- M. Dantas, D. Leitão, C. Correia, R. Macedo, W. Xu and J. Paulo, "MONARCH: Hierarchical Storage Management for Deep Learning Frameworks", 2021 IEEE International Conference on Cluster Computing, 2021.

- M. Dantas, D. Leitão, P. Cui, R. Macedo, X. Liu, W. Xu, J. Paulo, "Accelerating Deep Learning Training Through Storage Tiering", The 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, 2022.

Moreover, Monarch is publicly available as an open-source project at **`https://github.com/dsrhaslab/monarch`**.

1.5 Document structure

This document is structured in the following way: Chapter 2 elaborates on the state-of-the-art, where general background concepts are discussed, giving clarity on DL concepts that are related to this work, focusing more on the I/O-bottleneck of DL jobs, and how this problem is managed by both storage solutions and DL frameworks. Not only that, but relevant related work is also discussed, concerning developed systems that target DL training acceleration through I/O optimizations. Chapter 3 presents a preliminary experiment to validate the use of local storage to optimize DL training. Chapter 4 gives a detailed explanation of Monarch, followed with an extensive evaluation in Chapter 5. Finally, Chapter 6 will deliver this work's final conclusions and discuss some possible future research paths that build upon Monarch's contributions.

Chapter 2

STATE OF THE ART

Considering that this dissertation focuses on a storage solution designed for DL workloads it is necessary to understand fundamental DL concepts, specifically focusing on the DL training phase, its I/O requirements and how they are addressed by DL frameworks. Finally, details on how DL jobs are executed in HPC must also be provided to fully comprehend the problems that concern HPC storage infrastructure (*i.e.,* PFS). Apart from the necessary background, systems that target problems similar to those that motivate this dissertation are also analyzed.

2.1 Background

To better understand DL one must first understand ML, the field from which DL derives and shares base concepts, thus contributing to a better understanding of the cornerstone principles that sustain this work.

2.1.1 Machine Learning

Machine Learning (ML) comes from the simple idea of having a computer that can generate self-taught rules by looking at data. ML based systems should be automated by means of a training process, rather than being explicitly programmed, as in a more traditional and conservative manner, where data-processing rules must be previously defined.

The core of ML is the training phase, where many samples of a specific data domain (*i.e.,* images, text, audio) are fed to a model. The whole training process enables the ML model to recognize patterns and derive conclusions that will lead to the automation of a specific job, such as time-series forecasting, speech recognition and even autonomous driving. This process can be perceived as "learning" [27].

To sum up, the ability to have intelligence and automation, eventually, arises from the patterns, structure and rules that the ML model discovers while training with the provided data samples. As such, to achieve reasonable results, ML is dependent on the available training dataset. The model is often tuned to achieve a high degree of capability to generalize the results beyond the training data, so that it can be applied to data that it has not yet seen.

There are broadly four main branches of ML [27]. *Supervised learning* [43] where the model learns to translate input data to known labeled targets (*e.g,* object detection, image segmentation, regression). *Unsupervised learning* [42] is usually used as a first step towards *supervised learning*, in a way that it can help find and visualize correlations within a dataset without the help of labels. *Self-Supervised learning* [12] is *supervised learning* without labels provided by humans, but instead, typically using heuristic algorithms. *Reinforcement learning* [81] introduces the concept of *agent* as an entity that receives information from a certain *environment* and learns by being "rewarded" depending on the actions that it takes based on the received input. *Supervised learning* is the most common case of ML and it is the characteristics of this type of *learning* that this dissertation will focus.

2.1.2 Deep Learning

Deep Learning (DL) is a vast field inside of ML, where models are constructed upon several successive layers of connected *neurons*, which are the fundamental units of DL. By stacking each layer on top of each other a Artificial Neural Network (ANN) is created. This network can be broadly divided into three components, the *input layer*, the *output layer* and those that reside between the previous two that are called the *hidden layers*. When a ANN contains multiple *hidden layers* it is called a Deep Neural Network (DNN), as seen in Figure 1.

In a common ANN architecture each connection between the existing neurons holds a *weight*. The most commonly used types of neurons are the Threshold Logic Units (TLUs) [32]. These neurons compute a weighted sum based on the inputs they receive from their connections. An *activation function* is then applied to the sum value, which in turn, generates the output of that neuron. Some of the more well known *activation functions* are the Rectified Linear Unit (ReLU) [98] and the Hyperbolic Tangent (TanH) [97].

The training phase, similar to the one performed on ML models, happens via the *training loop*, where for each *iteration* of the loop, training samples and their corresponding targets are fetched from storage. This process goes on until the full training dataset is passed over multiple times. Each full dataset pass is an *epoch*.

Training samples are fed into the model, going from the *input layer* to the *hidden layers*, until they reach the *output layer*, to achieve predictions for those inputs. This process is called *forward pass*. As shown in Figure 2, having obtained the predictions, a *loss function* is used to measure how well the model is doing, calculating the *loss score*, which is a measure of the *error* between the predicted values and

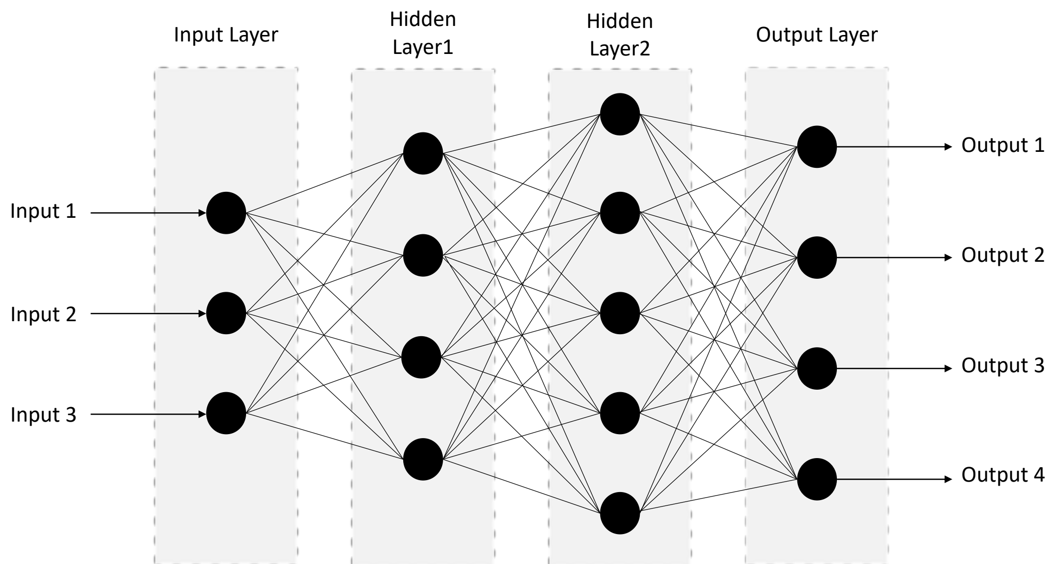


Figure 1: Deep Learning layers representation.

the real targets/desired output. The *optimizer* will then take the *loss score* and apply the *backpropagation algorithm*, going from the *output layers* to the *input layers*, to calculate the contribution (*i.e, error gradients*) of each connection of the layers to the *loss score*. With the *error gradients* of each connection's *weight* the *optimizer* will make a *weight update*, shifting the values of the *weights* in a manner that leads to the reduction of the *loss score*, targeting a *global minimum*.

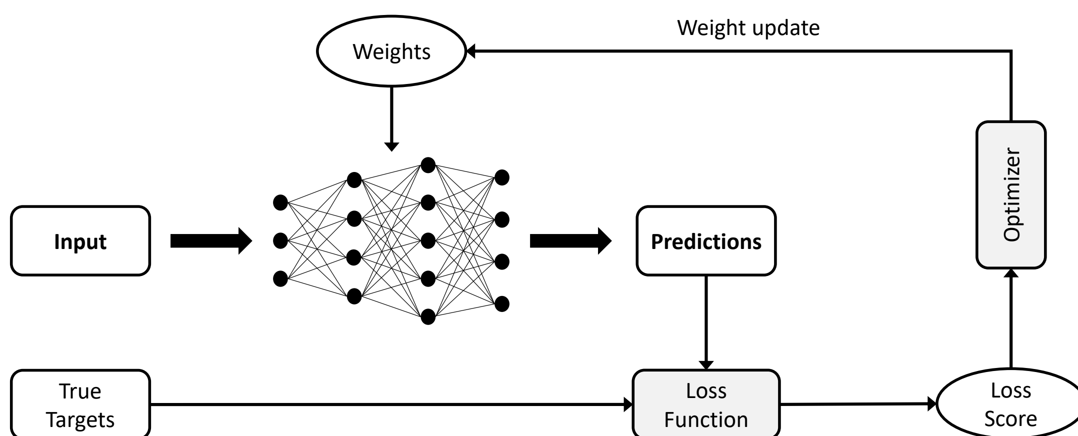


Figure 2: Deep Learning simplified training process steps.

Mini-batch GD is a popular algorithm for performing optimization on neural networks, where a *weight update* occurs after processing some predefined number of samples (*i.e.,* a *mini-batch* of samples, as seen in Figure 3).

With Mini-batch GD, each input that forms the *mini-batch* is usually picked randomly (*i.e.,* randomness

derived from the Stochastic Gradient Descent (SGD)). This randomness has benefits to avoid *local minima* and to find the *global minimum*. To achieve randomness, it is ideal to shuffle the whole training dataset in each *epoch*. This step must be done with the input samples and labels jointly, so that it does not interfere with the samples' labeling. It is important to make sure that each sample has the same *probability distribution* as the other samples and that all of them are statistically independent. This is done by either using a *global shuffling* method on the whole dataset (*i.e,* usually done by shuffling the names of the files to be read for each *epoch*) and then making storage requests that follow that predefined order, or by picking each sample randomly from the dataset [32] at the time of the storage request. There are discussions on how this random picking should be done [57], but it is certain that some degree of randomness is required and that *global shuffling* has convergence guarantees (*i.e.,* the guarantee that the model is approaching the *global minimum*).

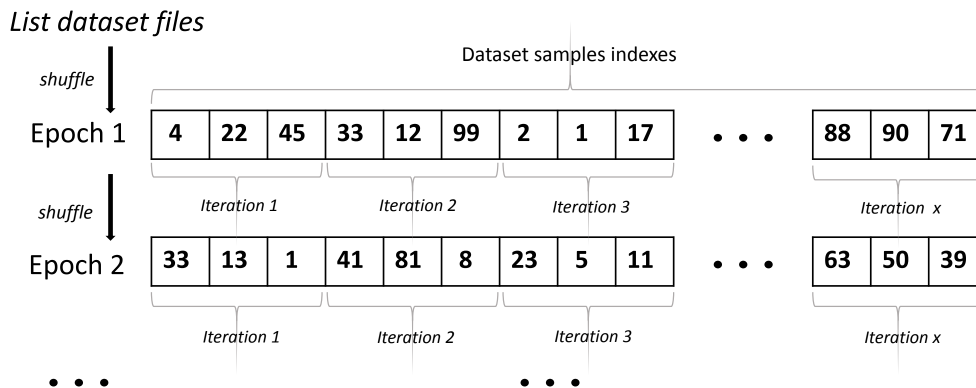


Figure 3: Example of the dataset access of a training loop that uses Mini-batch GD and *global shuffling*.

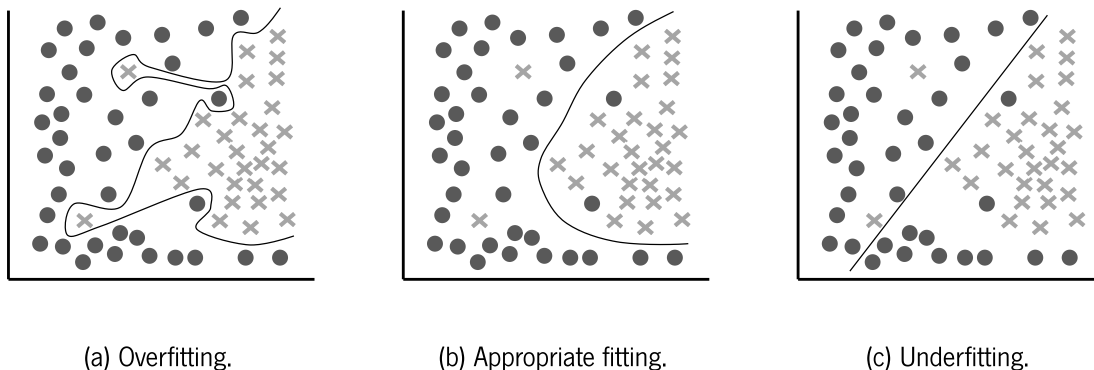
2.1.3 Overfitting

A common ML and, consequently, DL problem is *overfitting* the model to the available training data. When dealing with *i)* complex models, which in the case of DNN they usually are, and *ii)* training datasets that have noise or are not large enough to attenuate the sampling noise, DL models will find underlying patterns in the noise and will not generalize well to unseen data, thus failing at problem solving (*e.g.,* classify objects, speech recognition).

Even though regularization techniques can be applied during DL training (*e.g.,* l_1 and l_2 regularization [60], *Dropout* [79], *early stopping* [95]), very commonly the best solution for *overfitting* (Figure 4a) is to simply use a well balanced, diverse and large enough dataset [27, 82]. Techniques like *data augmentation* to artificially enrich the dataset with new data derived from existing samples can also be part of the solution for a better model [67].

The *stochastic* (*i.e.,* random) property of the Mini-batch GD can also be helpful to prevent the model from being biased by the noise of a meaningful input order. The obstacle of biased data is even more evident for classification problems where the dataset is usually organized in a class/target manner. In this case, when fetching a *mini-batch*, a representative set of samples of the whole dataset is needed and not just of some specific classes/targets. Therefore, *shuffling* the dataset is common when this is the case, making this step important not only for convergence, but also to achieve good generalization levels.

Nevertheless, *underfitting* a model (Figure 4c) is also a possibility. This is the opposite of *overfitting*. In this case the model, after training, could not capture the relationship of the data that is being fed to it, thus lacking complexity and performing poorly. This is less common than *overfitting* and can arise from trying to solve the latter problem. The solutions to this problem is to simply make the model more complex or reduce restraints to the model such as *regularization*, if they are present.



(a) Overfitting. (b) Appropriate fitting. (c) Underfitting.

Figure 4: Overfitting, appropriate fitting and underfitting for a classification problem.

With all of this in mind, it is now easy to understand why large quantities of data must be used in DL training. Not only that but, the random access pattern, characteristic of the DL training phase, is also necessary for convergence guaranties and to help preventing overfitting.

2.1.4 Model's evaluation metrics

DL developers need model *evaluation metrics* to quantify the model's performance on solving a given problem. For example, in classification tasks that use a unskewed dataset (i.e., a dataset where the frequency of some classes is not bigger than others), a frequently used metric is *accuracy*. This metric focuses on the ratio between the total number of correct predictions *versus* the total number of predictions made. The metrics *precision* and *recall* are more often used in the presence of skewed datasets. When chosen correctly, these metrics are extremely useful. On an DL production pipeline there are essentially two phases of evaluation: *model testing* and *training validation*.

Model testing is used to determine how well the designed solution will generalize to new cases (*i.e.,* data samples that were not used during the training phase), using a *test dataset*. This is a crucial phase that

must happen before the model's deployment, making it possible to know how well a model will behave on new inputs. The *training validation* phase is mainly used for model optimization, targeting the shift of the model's *hyperparameters* (*i.e,* properties that govern the entire training process), also known as *hyperparameter search*, and happens during the training phase. Tuning these properties is the process of trying to determine the model that obtains the highest *score*, utilizing a predefined *metric*, such as *accuracy*. When considering large datasets a *hold-out validation dataset* is used for this matter.

2.1.5 Deep Learning Frameworks

In order to facilitate the scientific advances of the DL field, building DL models cannot become very time consuming. With this goal, specialized frameworks were built to create and train DL models. These frameworks can be interfaces or libraries that help DL developers.

Firstly, DL frameworks contribute to the research and production of DL models, since they allow the fine tuning of the whole training process with the existence of low-level interfaces. These interfaces offer a high degree of customization and configurability to the internal processes of the framework, thus allowing DL experts the possibility to achieve high-performing specialized solutions. This is the case of TensorFlow, which is an open source software library, PyTorch, that can generally be found as a Python package, and MXNet, a framework specifically built to train and deploy DNNs.

Finally, these frameworks can also give non-experts on the underlying DL's algorithms and mechanisms (*e.g.,* Mini-batch Gradient Descent (Mini-batch GD)) the access to high-level interfaces. With this, users that are still new to the field are able to construct complex problem solving models. A framework that is well known to be *user-friendly* is Keras [41]. This framework focuses on offering a high-level Application Programming Interface (API) to build DL scripts. It runs on top of other DL libraries, more predominantly on top of TensorFlow.

This dissertation uses as use-cases the TensorFlow and PyTorch DL frameworks as these are two of the most popular frameworks currently available.

2.1.6 Deep Learning Execution Time overview

For this work, the total execution time of a given DL training job is based on DL frameworks' general *modus operandi* and will be divided into two core components: *computation time* and *data I/O time*. It is worth to mention that when parallel training execution is considered the *communication time* must also be added to the sum. The *communication time* is introduced by the weight movement costs between the devices involved in the distributed training. In the case of the *data parallel* model it is the cost of the all-reduce operation. Although important, this component will be set aside, since this work will be focused on *single-node training*.

The *computation time* in DL training includes operations such as convolution or element-wise arithmetic. The computations of a DL model are generally associated to the work of the *optimizer* (*e.g,* back-propagation and Mini-batch GD), including, obviously, the *forward pass* of each *mini-batch*. This time component is dependent on many hyperparameters, such as the number of *epochs*, *batch* size, layers complexity, *hidden layers* depths, etc. The higher the *epoch* size, the longer the training process and more full dataset traversals will be made. More *hidden layers* tend to lead to a larger number of parameters and a bigger model complexity, which also increases computation time.

Furthermore, DL workloads are *read-heavy*, but some write operations may occur over the training course, which result from *checkpointing* (*i.e.,* persisting) the DL model's state. These operations are done in files that are disjoint from the training dataset, as such, this dataset remains as *read-only* for the duration of the training phase. *Checkpoiting* provides some degree of *fault tolerance* in the presence of prolonged training times and *epochs*. However, for this work, *data I/O time* will solely be defined as the time that takes to fetch samples from the source dataset (*i.e.,* storage I/O) and deliver them to the model's *input layer*, thus ignoring the small percentage of I/O that comes from *checkpointing*.

Between arriving from the *storage backend* to being delivered to the model's *input layer*, data samples are usually *decoded* and can be preprocessed in memory, and in a wide range of ways. Some common preprocessing steps are *data normalization* and *data augmentation*, where, for example, horizontal flips and crops on an image can be made to benefit the model's performance [67]. Arguably this process can be considered a very distinct component of the DL training phase, but for the purpose of this dissertation it will be accounted as part of the *data I/O time*.

Although not being strictly imposed, the use of GPUs is widely adopted to reduce the computation costs. However, since the main computation is placed on the GPU and the data is in the CPU's memory, the data must be transferred from CPU to GPU (*i.e.,* involves traffic on the CPU-GPU interconnect), adding an extra component to the *data I/O time*. Notably, the preprocessing of data is usualy made on the CPU, but with the help of specialized data loading libraries, such as DALI [61], this can be made in GPU. Therefore, the *data I/O time* is mainly dependent on the throughput of the *storage backend* and the complexity of the preprocessing done to each sample (see Figure 5).

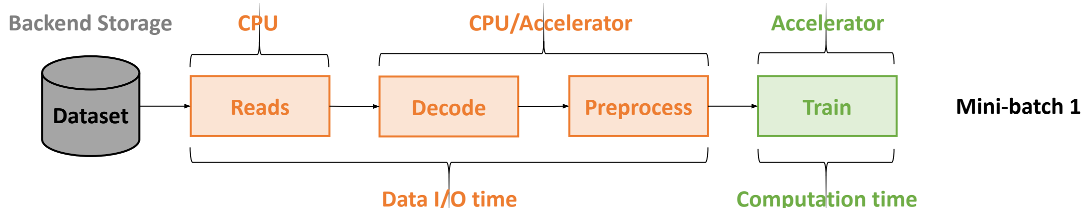


Figure 5: Deep Learning generic data-flow and execution time components.

With the help of DL frameworks, some of these operations can be overlapped, for example, computation can proceed while new inputs can be fetched, buffered and preprocessed to later on be passed to the model

input layers. These optimizations are often implemented in a pipelined manner.

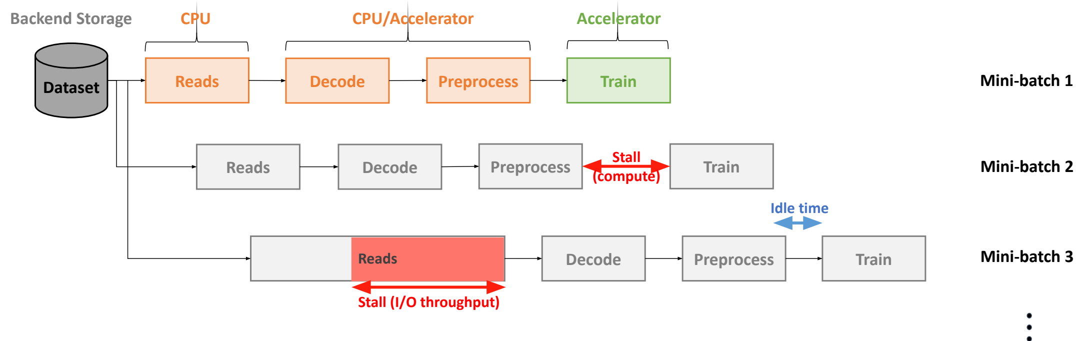


Figure 6: Deep Learning generic pipelined data-flow

A key concept to retain is that the predominant time consuming component will dictate where the performance bottleneck will reside. In Figure 6 *mini-batch 2* is ready to compute even before *mini-batch 1* training *step* is concluded, hence there is a compute stall that prevents *mini-batch 2* from being used for training. If for the majority of the training process data samples are always ready for computation and the *computation time* is predominant, then the overall configuration of the whole training process (*i,e. model's architecture, model's hyperparameters, I/O optimizations, hardware, etc.*) leads that job to be *compute-bound*. On the contrary, if for the majority of training *steps* the training loop sits idle waiting for input data, such as the case of *mini-batch 3*, where the I/O throughput was decreased for some reason and consequently delayed the training process, then the job will be considered *I/O-bound*. Generally speaking, DL jobs are usually *compute-bound*, but they can easily become *I/O-bound*, if no I/O-optimizations are used and under-performing storage backends are present.

2.1.7 Deep Learning on HPC systems

The performance of DL applications has been analyzed by many researchers and recent studies have shown that, within the *data I/O time*, storage access to read data samples cannot be dismissed as an existing bottleneck to the overall training efficiency [31, 63, 69, 92], especially in HPC infrastructures and their underlying PFSs, which are the main scope of this dissertation.

As stated in Section 2.1.3, DL training is associated with complex models and large-scale datasets. This leads to computational, network and storage needs for DL applications. Associated with these needs there has been an increase in the popularity of modern supercomputers. Modern HPC infrastructures, by design, provide parallel computing capabilities, enabling the execution of a particular job to be scaled up and breaked down into separate computational tasks to be performed by many individual cores within a single device or node. These systems also contribute to a scale-out job parallelism, where a job can be split into many parts that can be processed in parallel by different servers, this is particularly useful for the case of distributed training.

Storage access is easily obtained in these infrastructures by means of a shared PFS (*e.g,* Lustre, BeeGFS, GPFS), that provides fast global access to large volumes of data and ensures data persistence through a high number of distributed storage devices. Several supercomputers, such as Frontera and ABCI [5], also have their compute nodes equipped with local storage mediums, such as SSDs.

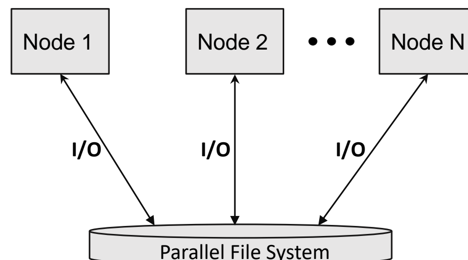


Figure 7: Parallel File System being used by multiple nodes.

Despite not being always necessary to use, since local storage mediums are also available, the PFS storage backend is appealing to users and they opt to store their datasets in that system for various reasons [24, 100]:

- Users might not be aware of local storage mediums available at compute nodes and their performance benefits
- In many cases, data must be manually copied from the PFS to local storage
- Large datasets may not fit entirely at the local storage resources
- Due to its shared namespace, data access in distributed training is guaranteed

Therefore, it is easy to understand that HPC users very often do not use the local storage that is available at the compute nodes. This generates a problem, since the PFS itself cannot handle very well DL workloads generated by the execution DL jobs.

This problem is evidenced by Chowdhury et al. [17], stating that DL workloads, impose serious challenges for PFSs and, in turn, lead to poor I/O performance of DL jobs. Traditional file systems are optimized for large sequential reads and under-perform in the presence of large datasets comprised of small files [51, 75] and random access patterns. This is also true for remote storage, such as a PFS. These systems offer large amounts of aggregate bandwidth, but when only small data requests are issued, only a small fraction of that bandwidth is used.

Moreover, in an analysis of the data stalls that occur in DL training, Mohan et al. [59], concludes that, when the dataset cannot be fully cached in memory, storage access can become a bottleneck, especially when accessing remote storage and not local storage. This study points out that implicit mechanisms, more specifically the *Page Cache*, are not efficient for DL training due to its *caching replacement policy*, that is

a variant of Least Recently Used (LRU) [88]. This policy, when the *Page Cache* reaches its full capacity, decides which cached items will be evicted to leave space for new insertions. Since DL training involves a repetitive and very often random access to data samples (Section 2.1.2), using LRU causes *thrashing*, leading to unnecessary item *replacements* and I/O movement. Figure 8 shows a simple example of the behavior of this type of cache. As seen in this figure, where the initial state is derived from a warmup phase (*e.g.,* after 2 training *steps*) and cached samples are continuously accessed in a random order, the *Page Cache* has already hit the maximum quota value and the replacement of items is active. In fig. 8, items that are cached are also evicted without being read by the application (*i.e., cache hit*). For example, item *C* is inserted in the first *epoch* of this figure, but in the second *epoch* it has already been replaced, which happens for the majority of items in this illustration. As such, this represents a waste of computational resources, with the needless replacement of items, increasing the number of *cache misses*.

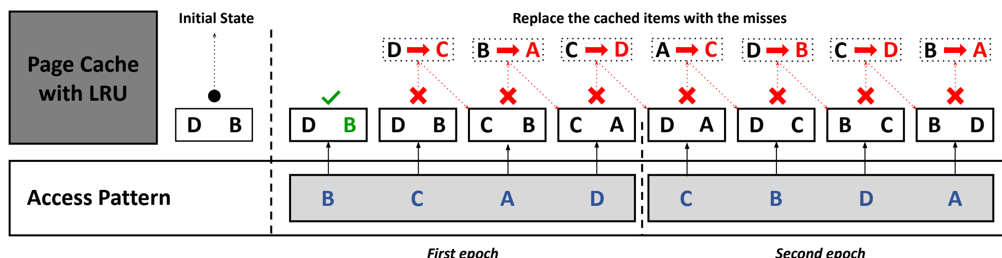


Figure 8: Simplified example of cache *thrashing* originated by the LRU replacement policy existent in the Page Cache, considering two epochs, cache quota of 2 units, and a worst case scenario for a random access pattern.

The size of a common deep learning dataset can easily surpass the caching capacities of a single compute node of a supercomputer (*e.g.,* Piz Daint [19] has 64 GiB of memory *per node* and Fugaku [73] only has 32 GiB). This is more evident when local training is used and a single node has to fetch all of the dataset samples. The lack of a better *caching replacement policy* is a reasonable concern since some DL frameworks rely on this caching mechanism to cache the data samples in memory and accelerate training performance (*e.g.,* PyTorch, which does not have any caching mechanism). Furthermore, HPC system administrators, recognize the I/O problem and try to obtain holistic performance guarantees for all users, but they can either rely on this mechanism to accelerate job's running times, and most importantly to minimize the number of requests issued to the shared PFS, or they must ask the users to have good practices [28], which is not an effective solution.

Wang et al. [92], took into consideration an AI specialized cluster, and showed that the I/O bottleneck is present across multiple scenarios, being predominant in *single-node* scenarios, whereas in a distributed setting it can potentially become the main performance bottleneck, but only when the *communication time* stops being the leading cause of performance deterioration, through optimizations.

Han et al. [31] empirically showed, in an HPC system with IBMPOWER architecture, that one of the problems of training on large datasets in HPC systems is the usage of a distributed file system, Lustre [77] in this case. This research compares the use of storing a dataset on a NVMe *versus* storing in Lustre. It proves that, owning to the fact that the file system performance depends on the number of I/O requests from all users and their diverse workloads, the performance variability using Lustre is bigger than that of the NVMe, which can be minimal. This leads to a boost in performance when using NVMe for models where the computational overhead is minimal. This is also backed by the common knowledge that shared storage systems like Lustre suffer from performance variability [53, 54, 96].

A study characterizing TensorFlow's I/O was also made [15]. In its experiments, it was observed that the delay originated by I/O operations can be completely hidden by prefetching, making the execution time non dependent on the number of threads used for data I/O and the storage technology used. On the other hand, threading can be used to increase the rate of file ingestion, but this effect is more noticeable on fast storage devices such as SSD, whereas in Hard Disk Drive (HDD), for example, the scaling flattens on a reduced number of threads. This gives the hint of the importance of using a fast local storage medium to store training datasets, instead of relying on a shared PFS.

2.1.8 Storage Solutions for Deep Learning

To speed up DL applications storage access times, some generic storage solutions might be used in combination with DL frameworks. In-memory Key-Value (KV) systems are used for DL, since accessing data through keys (*e.g.,* file name) is already the standard for DL frameworks and relational operations offered by Structured Query Language (SQL) databases are, generally, not needed. One of those systems is Lightning Memory-Mapped Database (LMDB) [18], a database based on a B+tree, exposing the entire dataset in an in-memory map, used by the Caffe DL framework [38].

Optimized data formats (*e.g.,* TFRecord [86] and RecordIO [72]) can increase I/O throughput by allowing large sequential reads and by drastically reducing the number of metadata operations issued during training, yet there are additional concerns when using these formats, provided by DL frameworks. Firstly, DL datasets are commonly found in their raw state (*i.e.,* comprised of small files, possibly divided into multiple directories), hence a conversion to the optimized format is needed. Secondly, the codebase of the DL job needs to be changed, accordingly, which makes their adoption more difficult. Thirdly, since the data samples are converted to *records* (*i.e.,* data structure containing data and metadata from a dataset's sample) and serialized into single or multiple files, we can no longer perform random access to individual *records*, unless there is an extra step of indexing each *record*, thus knowing their exact location on the binary file (*i.e,* offset and size). This *indexation step* can be used in the RecordIO format and Figure 9 shows how *records*' IDs can be shuffled to enable random access to *records*, making *global shuffling* possible (2.1.2).

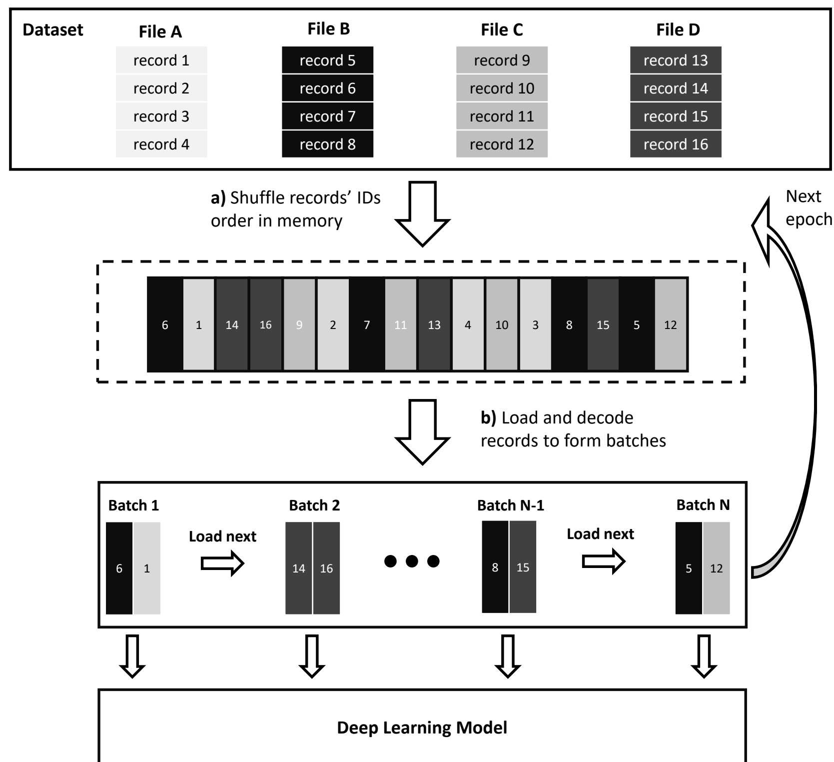


Figure 9: Random access to an optimized file format.

On the other hand, TensorFlow does not rely on random access to *records*, performing fully sequential reads instead. In this case, shuffling is provided by storing samples in a *shuffle buffer*, from where samples will be chosen later on. In addition and to increase the randomization level, the *IDs* of the files that hold the records can be shuffled, to obtain a different reading order in each epoch. Figure 10 explains this process, where the *shuffle buffer*, will be filled with samples in a sequential order and then records are randomly chosen from that buffer to form batches. State *1)* and *2)* represent possible states, since the buffering operation and the consumption of buffered *records* are asynchronous. The use of a *shuffle buffer* not only makes the randomization process dependent on the size of this staging area, but it further alters the provided level of randomization, which can lead to a worse model *accuracy* [57, 59]. It is worth to mention, that even though it is not illustrated in Figure 9 and Figure 10, the DL framework's loading process of data samples can also involve a *read buffe*r, orthogonal to the *shuffle buffer*, to serve as a staging area for data samples to be decoded and preprocessed.

Despite reducing metadata accesses, reading random *records* when a *file index* is present can defeat the purpose of having a binary format, not enabling sequential reads. Moreover, similar to TFRecords, when using the RecordIO format, a random chunk of *records* can be contiguously read from the storage backend, instead of a single *record*, reducing the problem.

Finally, it is important to note one inefficiency of optimized data formats. Going back to one of the problems discussed in the previous section, and as stated in [59]: "TFRecord format results in 40% higher

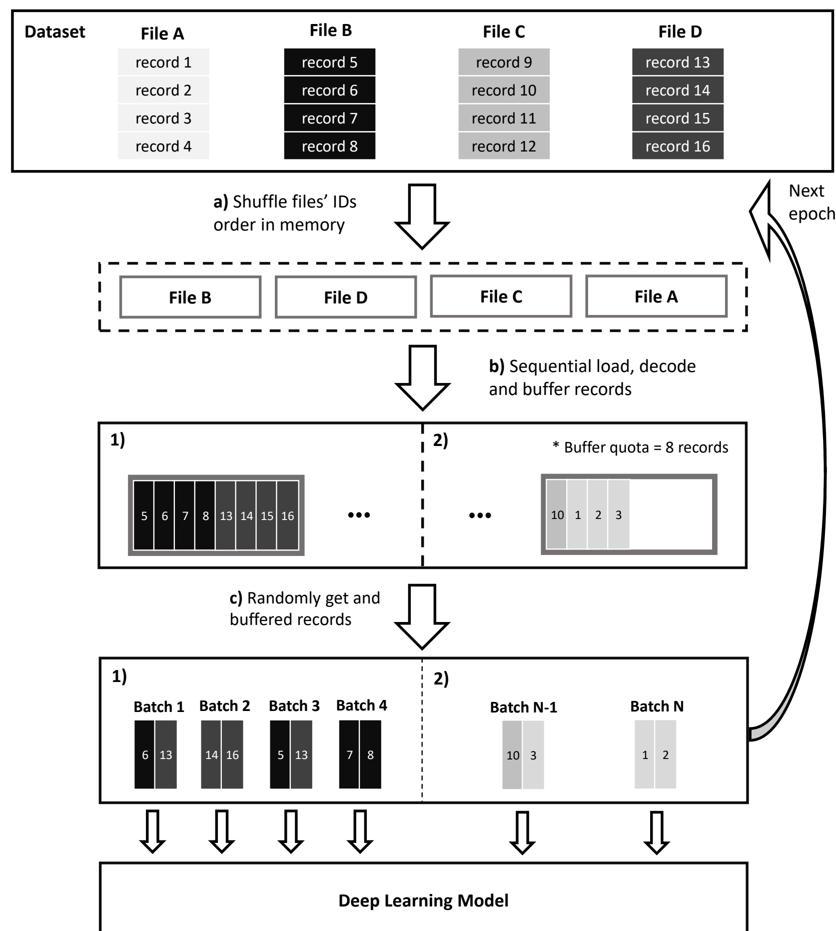


Figure 10: Sequential access to an optimized file format.

cache misses than the ideal because, the sequential access nature of TFRecords (and RecordIO) is at odds with LRU cache *replacement policy* of the Page Cache". In this case, using optimized data formats can increase cache misses in the presence of a transparent cache system that uses LRU as a replacement policy, when compared with DL training jobs that do not use these formats.

2.1.9 TensorFlow's Data Loading Solutions overview

To look at a different perspective on speeding DL training data I/O times the internal capabilities of frameworks can be analyzed. The TensorFlow framework's I/O operations are conveyed through the TensorFlow runtime, which is written in C++.

Through Python, TensorFlow provides the *tf.data.Dataset* API [85] that enables the creation of *input pipelines*. These pipelines typically perform I/O operations, such as reading, decoding and preprocessing data, and can be seen as *producers*, while accelerators, like GPUs, ingest their output, being the *consumers*. This API optimizes the training process by parallelizing data extraction and overlapping data I/O with computation, reducing accelerators idle time [15].

This framework provides specific *stages* to be applied over a data loading pipeline. Some of the most interesting *stages* to describe for the context of this dissertation are:

- ***Map*** is used to implement user-defined data stages, such as data preprocessing, over each sample that is passed to the model. This *stage* can be sequential or parallel, offering the possibility to accelerate data stage operations.
- ***Prefetch*** enables the overlap of data extraction with the model computation. In a broad sense, this *stage* uses a dedicated background thread to read samples from the *storage backend* before the time that they are needed by the *consumer*. This technique demands the existence of an internal buffer to store the prefetched samples. Since it prefetched indivudual samples, this is a *sample-based* prefetching optimization.
- ***Interleave*** can be used to parallelize data loading. For example, in a dataset composed of multiple input files that contain multiple samples (*e.g.,* TFRecord files) this operation, when used as *parallel interleave*, fetches a certain number of samples (defined by the *block_size*) from an input file until it moves on to another file. The number of input files processed concurrently is given by the *cycle_length*. This process can be seen in Figure 11, which can be related with the concepts discussed in Section 2.1.8 and Figure 10.
- ***Shuffle*** is used to allow the construction of a *shuffle buffer*, as described in the previous section and Figure 10.
- ***Cache*** is the *stage* that allows the caching of the dataset either on memory or on local storage, saving data and metadata operations from being executed to retrieve dataset inputs from the storage backend.

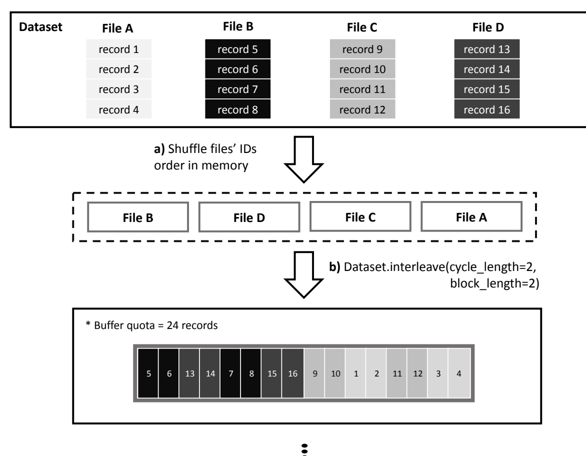


Figure 11: Interleaved access to an optimized file format.

All of these *stages* have configurable parameters, for example number of threads used (*num_parallel_calls*), and they can be either manually configured or left alone to be auto configured with *autotuning* algorithms (*tf.data.experimental.AUTOTUNE*), provided by the framework.

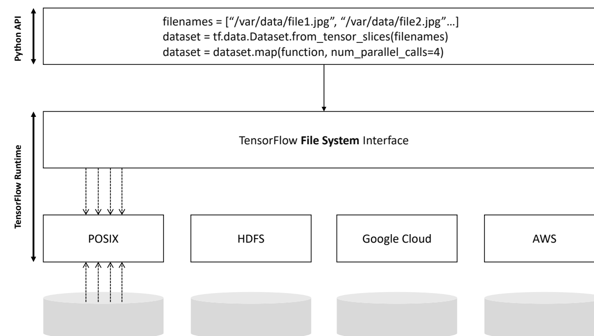


Figure 12: TensorFlow's file system adapters.

TensorFlow also assists the process of reading data samples from different sources, providing adapters to interact with various file systems, like Amazon S3, Google Cloud Storage, Hadoop Distributed File System (HFDS) and standard POSIX-compliant solutions, exposing the same interface to users, as presented in Figure 12.

2.1.10 PyTorch's Data Loading Solutions Overview

The PyTorch framework enables data loading optimizations and utilities through the *torch.utils.data* package [70]. More specifically, the *Dataloader* class is the heart of PyTorch data loading. This class will deliver the data samples to the training loop on demand.

The *Dataloader* can be enabled to use multiprocessing, and it does so by utilizing a main process interacting with spawned background worker processes, enabled through multiprocessing queues. The main process can submit requests to its workers to load data samples. *Prefetching* can be enabled by increasing the rate of requests that the main process issues to the worker processes. PyTorch's *Dataloader* background worker processes will simultaneously load and preprocess data samples, which causes an increase in data throughput to the training loop.

The *Dataloader* can also make data transfer to CUDA-enabled GPUs faster, by using *pinned memory buffers*. This technique is effective, since host to GPU copies are faster when they are originated from pinned (page-locked) memory [68].

PyTorch offers a considerable level of programmability for data loading and does it by designating the *Dataloader* class an iterable over a *Dataset*. The latter represents the class where the logic to fetch a data sample for a given key requested by the *Dataloader* must be programmed. On top of this, The *Dataloader* uses the *Sampler* to request samples from the *Dataset*. Both *Dataset* and *Sampler* are base classes, meaning that custom subclasses can be constructed on top of them, following a well-defined interface.

For example, the custom logic implemented on a *Dataset* can purely be to get data samples from a folder containing many subfolders with images at the root level. As for the *Sampler*, a classic custom strategy can simply be to enforce the random order of requests, in other words shuffling, or even partition the requests for a data parallel execution.

Although PyTorch provides some needed and broad implementations of these two classes, it typically entrusts the work of implementing custom logic to the programmer, for example, it does not have well defined adapters for reading data samples as in TensorFlow.

2.2 Related work

It is clear that storage I/O can become a bottleneck in DL training. This work will highlight many optimizations and storage solutions specifically designed to improve the I/O performance of DL applications workloads, that focus on different aspects of the problem, as well as more generic solutions that target scientific workloads in general, but that may also improve the DL training process.

2.2.1 Data Ingestion Pipeline

Some proposals improve DL frameworks' data loading and preprocessing efficiency by resorting to optimizations of different aspects of the data ingestion pipelines.

Prisma [55] proposes a Software-Defined Storage (SDS) [56] *data plane* that performs *file-level* prefetching to memory in a parallel manner. This system relies on enforcing the DL framework to read data samples using a predefined global shuffling order, that should be obtained by user interaction. This order is then shared with Prisma to apply efficient loading of the training data samples.

DALI [61] is a library designed to accelerate DL training, through optimized and transparent data loading techniques, such as parallel execution, *sample-level* prefetching (*i.e.,* prefetch individual samples and not whole files) and *batch* processing. It specializes in the loading and processing of image, video and audio data. It stands out by addressing the problem of the CPU bottleneck, by making it possible to do data processing, such as decoding, cropping and resizing in GPUs, allowing direct data path between storage and the GPU device. DALI is meant to be a direct replacement for built-in DL frameworks' dataloaders and iterators (Section 2.1.9 and Section 2.1.10).

The strategy proposed by Lanaras et al. [48] aims at improving GPU access to the storage system, focusing on providing a data path from storage to GPU. Firstly, it provides a Data Tracking Tool (DTT), which can be seen as a data iterator that is integrated with the Caffe DL framework. The DTT provides in each training iteration data pointers to the GPU, allocating memory on this device to prefetch *batches* of data. Not supporting online preprocessing stands as a major flaw for this iterator, thus it needs to access data that is already in its final state. Secondly, it provides a methodology to support GPUDirect Remote Direct

Memory Access (RDMA) [29]. The latter contribution could not be empirically tested, since at that time storage support for GPUDirect RDMA was not available, hence the DTT iterator was only tested by moving data from Random-Access Memory (RAM) to GPU.

2.2.2 Parallel I/O

I/O inefficiencies are found when using DL frameworks in combination with specific storage access formats that impair or reduce the effect of utilizing parallel I/O during the DL training phase.

Pumma et al. [69], with the LMDBIO plugin, optimizes Caffe's LMDB I/O subsystem to improve the mapping and caching of training data from storage to memory. This system tackles inefficiencies of using the `mmap` call and LMDB. LMDBIO, for example, diminishes the number of context switches and sleep time from executing `mmap` with many parallel processes. It performs this by defining a root process responsible for reading data and distributing it to the remaining processes with MPI shared memory. Furthermore, this I/O plugin also provides *speculative parallel I/O*, to surpass an LMDB inefficiency, when reading data records, that leads to redundant I/O. It also reduces the `mmap` workflow overhead by replacing that call with explicit I/O (*i.e,* the `pread` system call), among other optimizations.

TensorFlow allows for parallel processing of the training data samples with the *map* operator (see Section 2.1.9), but when certain Hierarchical Data Format version 5 (HDF5) libraries are utilized, by necessity and in detriment to the TensorFlow *input pipeline*, all operations can be serialized, thus negating any kind of parallel execution. Kurth et al. [47] uses the Python *multiprocessing* module to allow the parallel execution of worker processes, each one with a HDF5 library instance, to overcome this problem and optimize the data processing step.

2.2.3 I/O Buffering

I/O buffering is another solution to increase applications' performance, and it is widely utilized in a variety of forms in generic computational environments.

2.2.3.1 Burst Buffers

The concept of *burst buffers* is well known in the HPC community. These systems are very often composed by a tier of SSD devices, allowing to temporarily store and manage bursty I/O from HPC applications [52].

In a HPC infrastructure these systems can be used as a staging area, providing higher bandwidth than the PFS, leaving the latter to be provisioned for capacity and resilience. Later, if necessary, files staged at these storage layers can also be transferred to the PFS for persistence. Such is the case exposed by Cray DataWarp's design [49]. This storage system joins many SSDs to form a service that results in a intermediate storage layer (*i.e.,* in-between the compute nodes and the PFS) that, through user configuration,

dynamically provides a requested amount of storage space, noted as an *instance* that can be valid for the lifetime of a job or persistent.

DataWarp *instances* can be configured with two different types. With the *scratch* type, users' applications must explicitly move data between DataWarp and the PFS (*e.g.,* stage additional files, read/write data). This is achieved with the help of script-accessible and application-accessible APIs. Another type of *instance* is *cached* where data movement becomes implicit (*e.g.,* includes *read-ahead* and *write-behind* capabilities, and LRU tracking), and no explicit requests are required. The storage provided by DataWarp *instances* can also be shared by multiple jobs that run concurrently or sequentially.

Furthermore, DataWarp can be considered as a *remote burst buffer*, and can even be used as an I/O storage system for on-demand deployments of PFS systems, such as BeeGFS. Systems like IME [35] and aBBa [26] are other examples that also follow the design principle of *remote burst buffers*. However, this design fails to properly utilize the already existing local resources of many HPC systems, and relies on a new tier of devices to be deployed on, as well as needing an experienced programmer to utilize them correctly.

In contrast with the latter *burst buffer* design, *local burst buffers* are also being discussed and examined by the scientific community. BurstFS [93] is designed to support the aggregation of I/O across distributed node-local storage for the same lifetime of a job. When a *batch* job is allocated to a set of compute nodes on an HPC system an instance of BurstFS is constructed using local resources, such as SSDs or memory. This is achieved by mounting this system with a configurable prefix, by transparently intercepting POSIX functions targeted to that prefix and by utilizing a distributed key-value store for metadata management. However, BurstFS design is targeted towards checkpoint/restart and multi-dimensional I/O access workloads (*i.e.,* multi-dimensional variables are written in one particular order, and can be read for analysis or visualization in a different order than the write order), which are different from DL workloads.

GekkoFS [90] presents a local, temporarily deployed and distributed file system, providing a global namespace that is accessible by all participant nodes. This system allows for POSIX relaxation, by not providing global locking mechanisms, leaving to the application the responsibility of making sure that no conflicts occur. GekkoFS uses a pseudo-random distribution to spread data and metadata across all involved nodes. This system has two primary components, the first is a client library, that needs to be preloaded by the running application, making it possible to intercept of all related file system operations to be forwarded to the second component, which is a server process/daemon to handle the requests and serve them accordingly. This system, although transparent, relies on the explicit behavior of the application, not providing any kind of data placement when the samples are stored at the PFS, unless it is demanded.

Following this trend, Kung et al. [83] optimizes the local design of *burst buffers*, by showing that RAM can be used to solve the impact on the draining speed of distributed node-local *burst buffers* to the PFS (*i.e.,* to persist staged data), when their capacity is exhausted during high I/O peaks. This work exposes *proactive draining*, where data when available at the *burst buffer* is divided into small blocks and write

requests to the PFS are dispersed evenly across the entire computation and I/O period of the application.

2.2.3.2 File System Optimizations

Some relevant works focus on optimizing existing file system solutions or on providing new paradigms to rethink file systems for the exascale era.

FS-Cache [34] is a kernel facility for Linux by which data retrieved from over the network (*e.g.,* from a PFS) can be persistently cached locally, trading disk space to gain performance improvements. This system can handle partially cached files that do not fit in the local predefined cache. FS-Cache is applicable to single-node scenarios and needs a cache backend to properly function. A cache backend is a storage driver that needs to be mounted and configured to provide caching services.

LPCC [71] integrates with the Lustre's Hierarchical Storage Management (HSM) solution and the Lustre's layout lock mechanisms to couple nodes' local SSDs with the PFS and its global namespace. LPCC provides a persistent caching system, by using the HSM mechanisms for data synchronization and can provide either a read-write cache for single clients or a read-only cache for multiple clients, using the local SSD. This system also provides a prefetching mechanism that follows rules or hints configured by the user. LPCC can be configured to have different caching eviction policies to apply in the presence of cache saturation, such as LRU or even a no-eviction policy. LPCC is specifically designed to work with the Lustre PFS, however other PFS systems have HSM mechanisms implemented, so this system can be generalized to other systems, but the specific implementation details have to be adapted.

Differently from the previous file system optimizations, DeltaFS's [101] states that today's PFSs continue to feature outdated semantics, such as having their persistent state globally synchronized at all times. Thus, DeltaFS provides a relaxation of the file system's namespace synchronization and serialization. It does this with the help of *per job* metadata log records, used for an application to register the namespace changes that result from its execution. These metadata logs do not need to be constantly merged back into a single consistent global namespace, instead, a job can selectively merge metadata logs produced by previous jobs to form new file system's namespace views (*i.e.,* a namespace snapshot) for sequential data sharing. DeltaFS also allows jobs to self-manage their synchronization scopes to improve performance.and removes the necessity of having to dedicate a single metadata service to meet the needs of all applications. Moreover, this system requires an underlying object storage service to store file system's metadata and file data. In addition, a *registry daemons* that runs on dedicated server nodes for inter-job communication (*i.e.,* sharing snapshots) is also needed. DeltaFS can provide, upon specific configuration, the same semantics as *local burst buffers* to provide ephemeral namespaces or persistent ones. Furthermore, each job can start its own DeltaFS metadata server processes on its own compute nodes to perform namespace merges and then to serve the reads, making use of local resources.

2.2.3.3 Deep Learning Focused Caching

There are many systems that follow some *burst buffers* principles, such as, offloading I/O from the PFS to improve applications performance, but focus specifically on DL applications running on HPC infrastructures.

Fanstore [100] aggregates the local storage of several compute nodes to enable data sharing in distributed training environments by providing remote file access with MPI and metadata broadcast across the participating nodes to maintain a global namespace. Fanstore can be used transparently by DL users and in a transient manner, but it requires the full dataset to fit in the node's aggregate distributed storage, hence it imposes distribution on the training process. Furthermore this system requires an extra data preparation step before training, where a user must provide a list of all files that will be involved in the training process to a preparation program.

Diesel [91] resorts to local storage mediums and an external distributed key-value store service to cache data and metadata information. This system allows users to convert and aggregate small files and their metadata into larger chunks of data. It also uses a *chunk-wise shuffle* to improve the performance of reading small files. This operation works similarly to the TensorFlow loading and shuffling mechanism for TFRecords (Section 2.1.8) and has the same issues.

CoorDL [59] provides insights on storage I/O data stalls and mitigates them by providing a specialized in-memory *caching replacement policy*. The solution is targeted for the DL access pattern. In CoorDL, with a single node training scenario, data samples, once cached are never evicted again, and once the cache fills the DL framework requests are directed to the default storage backend. This policy avoids *thrashing*, since it is not important which data is cached, since all data samples present the same probability of being accessed across all DL training *epochs*. CoorDL also provides optimizations for distributed training. One of them being *partitioned caching*, that allows remote file access between nodes over Transmission Control Protocol (TCP). In addition it also provides *coordinated prep* that accomplishes the re-utilization of already processed batches of samples to be staged and then shared among synchronized DL jobs that are accessing the same dataset to perform hyperparameter search (Section 2.1.4). It is worth mentioning that this system needs to be integrated with a DL framework, by being a direct replacement for the framework data loading mechanisms.

Serizawa and Tatebe [78] focused on data staging utilizing the compute nodes' local storage (*e.g.,* SSDs) with the intent of optimizing DL training performance. Their approach establishes the principal goal of concealing the copy of the dataset to the local storage with a pipelined solution. The first step of the designed pipeline is to generate a list of indexes that define the data samples that will be present in each *mini-batch* (Section 2.1.2). The second step is to share that list with parallel worker processes, so that they can prefetch each mini-batch (*e.g.,* from the PFS) and stage them at the node's local storage. The final step is to read the staged samples to form the corresponding mini-batch.

The previous proposal follows the design of a specific data loading class in the Chainer DL framework [87] and its logic is completely merged and integrated in it, which leads to a platform specific solution. This work fails to properly utilize the local resources of compute nodes, not addressing the fact that the dataset may not fit entirely on local storage.

AIStore [6] supports data processing pipelines that can execute on storage nodes and/or compute nodes, inserting tensors directly into the GPU memory, through RDMA. AIStore provides a scalable namespace over a arbitrary number of disks, having the data flowing directly between compute clients and clustered storage targets, providing a Representational State Transfer (REST) interface for clients. It uses Hypertext Transfer Protocol (HTTP) redirects to enhance storage access control. Parallel dataset re-sharding is additionally used in order to improve I/O performance, by aggregating small files into bigger shards, improving performance with larger reads. The AIStore solution was only analysed in PyTorch by implementing WebDataset, extending PyTorch's Dataset class.

Finally, Deep Learning File System (DLFS) [103] builds on the idea of *storage disaggregation* to support DNNs, by allocating a collection of local and/or remote storage devices to serve as staging areas and improve the performance of DL jobs. By using Storage Performance Development Kit (SPDK)-based user-level NVMe over Fabrics, DLFS can use RDMA in the SPDK protocol to allow data on an NVMe SSD device to be accessible to all participating remote clients, which is ideal for distributed training. This systems performs a initialization step to build the connection between the allocated NVMes devices, stage the DL training dataset and build an in-memory sample directory to maintain a global namespace to track the location of data.

DLFS allows *opportunistic batching* which comes in two forms of optimizations. The first is *frontend sample-level batching* that translates to the sample-level prefetching, which is only possible by having a predetermined global list with the data samples' access order. The sample directory, through index management at the sample level will allow DLFS, similarly to the RecordIO method (Section 2.1.7), to have random access to any sample in a RecordIO or TFRecord file, which allows full randomization when using optimized data formats. The second optimization is to perform *bakend chunk-level batching*, that aggregates small data samples into chunks in the initialization process enabling larger reads, however it adds the necessity of a *shuffle buffer* (Section 2.1.7). This system, however, requires some degree of user interaction by enforcing the application to use a front-end API to mount the system and utilize its capabilities.

2.2.3.4 Data Substitution

Other solutions, besides providing I/O buffering capabilities, explore DL semantics to employ data substitution techniques where training samples being served to DL frameworks are replaced by others (*e.g.,* cached samples) that are faster to access.

Data echoing [16] is a simple optimization in the form of a *stage* for TensorFlow's tf.data API (Section 2.1.9). It tries to reduce the total computation used by earlier training pipeline *stages*, such as disk I/O and data preprocessing. Therefore, data echoing is designed to increase training performance in DL training phases where the computation time in the accelerator represents a narrow portion of the wall time, leading to idle time in those devices (Section 2.1.6). It does so by reusing intermediate outputs from earlier pipeline *stages* up to a predefined number of times, for example using repeated data that is already provided and transforming it differently with data augmentation.

For this technique to be used the practitioner must identify the major I/O bottlenecks of the pipeline and insert the additional echoing *stage* after that bottleneck. Although data echoing shows interesting results it reduces the number of new unseen data samples required for training. More prominently, if the echoing step is introduced at lower levels of the input pipeline, it exhibits slightly worse performance when using the ImageNet dataset to train the ResNet-50 model. Therefore, this setup required more fresh samples (*i.e.,* that were not reused through caching) than what would be expected, leading to a more extensive training time to reach the target *accuracy*. The data echoing solution can be abruptly reflected as a caching system that supplies repeated data when a cache miss would take place. The amount of repeated data must be defined in the pipeline *stage*. Contrary to other less intrusive solutions, like storage systems that also offer data samples caching, this solution can store intermediate pipeline results, instead of just raw data.

Ohtsuji et al. [64] proposes a preliminary work that aims at monitoring the I/O requests of DL jobs and skips the requests that are bound to be delayed, giving alternative training data to the training process, in order to avoid the eventual tail latency of the underlying storage system.

DeepIO [102] is an in-memory storage system designed for large-scale DL training on HPC systems and specifically designed for the TensorFlow framework. It stores data samples using in-memory buffers to facilitate the generation of randomized *mini-batches* to be ingested by the training loop. DeepIO is a solution focused on solving the massive number of small random reads directed to the backend storage. When the DL model requires a specific sample order and the dataset exceeds the buffer capacity, DeepIO implements Entropy-aware Opportunistic Ordering (EOO). With this method servers independently choose which samples will make the next *mini-batch*, utilizing only the elements loaded in memory. This solution uses input pipelining to overlap disk I/O, when the dataset does not fit completely in memory, reducing the impact of the mini-batch construction for that case.

Quiver [46] is a distributed cache for DL jobs input samples. This storage management solution has the key objective of improving cache efficiency, sharing data across multiple jobs, or even multiple users and can dynamically prioritize cache allocation to jobs that benefit the most from caching, such as compute-bound models. Quiver also places a significant emphasis to security in data sharing, without data leakage and using secure content-based indexing of the cache. Quiver is not solely built for HPC infrastructures, but designed for a shared GPU cluster allocated in the cloud. However, it stands out by also using data sample substitution to avoid cache *thrashing*. It achieves this by replacing data samples that resulted in

cache misses with cached data that was not yet utilized in the ongoing epoch, consequently changing inputs order. For its caching strategy to work properly, Quiver must evict samples that were already used by all participants in each training epoch. Replacing cached samples enforces the need of having to fully read the dataset in each epoch from the source where the dataset is originally stored, which does not reduce I/O pressure at storage systems, like the supercomputer's PFS.

Even though it contributes to a clear boost in performance, systems with data substitution have to be handled carefully. Altering the order of the inputs provided to the framework can lead to a worse convergence of the model (Section 2.1.2). Choi et al. [16] clearly evidences that fresh data (*i.e,* data that was not used in the current *epoch*) is inevitably required and more of it leads to a higher *evaluation metric* (Section 2.1.4). Zhu et al. [102] (*i.e,* DeepIO) goes to show that their pipeline does not affect the delivered *accuracy*, however it implies that depending on the *mini-batch* size it might need some careful engineering work on the training parameters in order to maintain high levels of *accuracy*. As well as DeepIO, Quiver proves, using specific use cases, that their substitution technique does not affect the *accuracy* of the model, however these solutions are dependent on the buffer size to store enough amounts of data that enables the randomization process to still be adequate. When dealing with memory restricted environment this systems can cause harm to the model's final *evaluation metric*.

DL training metadata is usually fetched in separate from the actual data, unless an *optimized data format* is being used (Section 2.1.8). This metadata is extremely important for DL training. Taking the example of an image classification problem, the metadata provided to the DL framework is what enables the correct training of the DNN, by providing the classes of each image. Data substitution techniques can lead to more intrusive solutions, since this metadata needs to be joint with the corresponding data. By substituting data samples these systems also need to provide the correct metadata to the framework, thus an intrusive interface or *client* is needed to replace the DL framework's built-in data loading mechanism.

Finally, Yang et al. [94] provides caching capabilities in the form of distributed caching. In this solution, a so-called Locality-aware Data Loading (LDL) algorithm is enforced for distributed training, where learners can assemble a mini-batch from their locally cached data. This can also be seen as data substitution technique, but of a very different kind from the previous systems. In its algorithm, all samples specified for a given *global mini-batch* will still be used for each training step. However, the participating nodes, instead of reading samples in the predefined *local mini-batch* block order (*i.e.,* order of the index list, see Section 2.1.2), will read the samples that are found in their local cache. The samples that are read must still belong to the *global mini-batch*, hence the difference from the previous systems. Nevertheless, this leads to an imbalanced distribution of samples in each node as a result of the *global mini-batch* being randomly sampled. To counter this a load balancing phase is needed. In it, the participating nodes need to agree on how to load samples locally, so that they collectively assemble the *global mini-batch*. Learners can subsequently achieve load-balancing by delivering samples either from the storage system, if they are not in the cache, or by exchanging samples with other nodes.

This work, elaborates on a *proof of equivalence* on the results of using this technique, showing that the ordering of the samples within the *global batch* does not affect the training results after global synchronization [10]. However, this system can affect the very common technique of *batch normalization* [37], when the latter is applied to the local parts of the *batches* (*i.e.,* found in each participating node).

2.2.3.5 Storage Tiering

Finally, there is a range of systems that, although they can provide caching of some sort, are characterized especially by their ability to utilize multiple storage tiers (*e.g.,* compute node's local disk and memory, PFS) in an efficient manner.

NoPFS [24] uses a performance model to proactively fetch training samples, to different storage tiers (*e.g.,* RAM and node-local SSD) and distributed memory (*i.e.,* remote access to other nodes memory), before these are requested by the DL framework. NoPFS relies on a in-memory *staging buffer* to store samples that will be first consumed by the DL training phase. After being used, training samples are evicted to serve space for the next samples that are being prefetched. There is no eviction for the remaining storage tiers.

The NoPFS system is fully integrated with PyTorch and uses the fundamental idea that, by knowing the seed for the pseudorandom number generator that determines the samples access order, it is possible to know the sequence of samples that will be read by each worker process (*i.e.,* similar to [55] and [78]). With this, it is possible to make a probabilistic analysis of the predefined access pattern and show that there is almost always an imbalance in the frequency a worker node accesses a particular sample. Knowing which samples are more frequently accessed by each node will allow this system's performance model to cache more frequently read samples at faster local storage tiers to decrease DL training times. 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.

Hermes [44] provides a storage tiering solution for partially or totally buffering I/O from scientific workloads at intermediary local storage mediums and across remote nodes (*e.g.,* using RDMA). This systems offers both an intrusive mode, with an API that users can interact with, and a transparent mode to transparently intercept POSIX system calls. Hermes can be used in a *persistent buffering mode* where data that is buffered by this system is also persisted to the PFS. This mode is used mainly for write-oriented workloads that need persistence. Hermes can also be configured with the *non-persistent buffering mode*, which is a mode designed for fast temporary I/O, used mainly for storing intermediate results and to do in-situ data analysis and visualization. In addition, Hermes provides a wide variety of *data placement policies*. For example, the *maximum application bandwidth* policy starts to place data in the first layer (*e.g.,* RAM) and when that layer is full, it goes to the next one, and so one. The latter policy moves data down when the maximum amount of storage space is needed (*i.e.,* where data moves from RAM to SSD). This data movement can, however, become an eviction policy, when there is no space left in the lower

storage tiers. Another example of a *data placement policy* is *hot-data*, where buffering is provided for data that is frequently accessed. Moreover, Hermes allows a user to provide custom *data placement policies*. Besides offering I/O buffering features, Hermes has the ability to change the buffering schema (*i.e.,* the order to where data is placed) dynamically by monitoring the system status such as capacity of buffers and messaging traffic.

Similar to *burst buffers*, Hermes is designed for general purposed scientific workloads not having any default *data placement policy* designed for the I/O patterns of DL jobs (*e.g.*, full dataset is read for each training epoch, random I/O accesses, possible use of optimized file formats). For this reason, Hermes is complex to configure and demands that users know exactly what their workloads involve, which is not trivial.

Data Elevator [23] proposes to transparently and efficiently move data between storage layers in HPC systems with hierarchical storage available. The Data Elevator system, instead of having the user involved in data movement, intercepts and stages applications' write requests on a fast persistent storage layer, such as a layer of SSD-based *burst buffers*, for faster I/O. When the data placement is complete, applications can continue the normal course of computation, while Data Elevator asynchronously transfer that data to the PFS. Data Elevator evidences that *burst buffers* are bound by a fixed number of server nodes to perform data movement (*e.g.,* persist buffered I/O to the PFS), which will impose a physical bound on the I/O parallelism. Therefore, to reduce resource contention on the *burst buffer* layer, Data Elevator is instantiated either on the compute node were the application is executed or on a separate node. The asynchronous data transfer is done by reading the content written to the *burst buffer* layer to the compute node's memory, which, in turn, is transferred to the PFS. This work shows that reading data from the *burst buffer* layer to the compute node's memory is faster than writing data from the *burst buffer* layer to the PFS. Thus, the Data Elevator mechanism results in the reduction of contention at the *burst buffer* layer. Evidently, Data Elevator optimizations are solely focused on *write-intensive* workloads. Therefore, it is not useful for DL workloads.

2.3 Summary

Deep Learning (DL) frameworks support and hide most of the complex algorithms to build outstanding DL models, supporting both computing and data loading mechanisms. The execution time of the DL *training phase* will be dependent on the performance and behavior of those same mechanisms. DL training is associated with long training times, due to computational, network and storage needs. These needs lead DL users to use HPC infrastructures.

However, recent studies show that storage I/O cannot be dismissed as a bottleneck [15, 17, 31, 59, 63, 69, 92], increasing the I/O-bound aspect of DL models. HPC infrastructures users can suffer greatly from this problem, since it can lead to performance losses, but most importantly they can tarnish the overall

performance of the PFS. This stems from the unpreparedness of shared Parallel File Systems (PFSs) to support the random access pattern of DL applications, as well as the large amounts of data needed for the training phase, which are very often found in the state of small files. Furthermore, the datasets used in DL training become problematic, in terms of I/O performance, for transparent caching mechanisms used in compute nodes, such as the *Page Cache*, when a random storage access pattern is present. Therefore, DL jobs can suffer performance losses and their workloads can further increase performance variability of concurrent jobs [53, 54, 96] due to the additional strain caused by the immense data and metadata accesses that are directed towards the PFS. Nevertheless, since the PFS is a standard choice for storage access during DL training, the local nodes' resources, that could increase applications I/O throughput [15, 31], specially in single node scenarios, are left unused.

Due to the storage I/O bottleneck of the DL training phase, storage systems are used to optimize the access to data samples [18, 30, 72, 86]. Some of these systems aggregate small files into larger ones to reduce the number of metadata operations and perform sequential reads (*i.e.,* TFRecords). DL frameworks also acknowledge the storage I/O bottleneck and allow programmers to build optimized input pipelines in conjunction with optimized storage solutions, however the I/O bottleneck still remains.

Therefore, multiple DL targeted optimizations address the DL storage I/O bottleneck by optimizing the data loading pipeline [48, 55, 61], including data preprocessing and prefetching capabilities, as well as data transfers to GPU. Researchers also took notice that there is room for further parallel optimizations in regards to the already existing storage access utilities of certain frameworks [47, 69]. While this dissertation design leverages ideas from these works (*e.g.,* prefetching, 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 this work and can even be used in conjunction with it.

Likewise, I/O buffering solutions resort to local resources, such as SSD devices, as a mean to increase DL jobs performance. System designs that offload pressure from the HPC infrastructure's PFS such as *remote burst buffers* [26, 35, 49] or *local burst buffers* [83, 90, 93], are mainly targeting general write-oriented scientific workloads. In addition, file system optimizations exist [34, 71, 101], but these are targeted at generic workloads. Further, they need complex configuration steps, in order to be correctly deployed, and even require the use of a specific PFS [71].

Other works developed caching mechanisms specifically focused for DL [6, 59, 78, 91, 100, 103]. Nevertheless, the majority of these systems are designed for specific DL frameworks and are intrusive, requiring DL developers to make changes to their DL traning scripts [6, 59, 78, 103]. Coupled with this, not all solutions allow the partial buffering of a DL training dataset [78, 100]. Furthermore and although they are transparent, some designs require the allocation of additional resources and the orchestration of complex data and metadata staging areas [91, 100]. On the contrary, this dissertation focuses on a solution that is designed for single-node training and provides a self-contained middleware that avoids this,

while not assuming that the dataset fits entirely on faster storage tiers.

Also targeting DL workloads, besides offering I/O buffering, some system employ data substitution techniques [16, 46, 64, 94, 102], but these can effectively impact the *accuracy* of a model and need to be carefully handled. Furthermore they are intrusive to the DL framework. Most of these solutions are useful for scenarios where several jobs are training models from the same dataset (*i.e.,* shared dataset). Differently, this work optimizations are designed for single-node training scenarios where, to improve the model's *evaluation metric*, each file of the dataset must be read once per epoch.

Finally, storage systems that leverage different *storage tiers* must be taken into consideration, but they can also be intrusive and do not provide modular optimizations [24]. This dissertation targets storage tiering, but it outsources the proactive data fetching to built-in mechanisms already present in DL frameworks, or provided by external solutions such as DALI. Other systems are designed for general purpose jobs [23, 44], being specially optimized for *write-intensive* workloads. This dissertation proposes a solution that is targeted towards read-oriented DL training workloads that change the way data samples must be placed across storage tiers.

With the existing related work in mind, it is crucial to build a framework-agnostic storage middleware that can be integrated with different DL frameworks without the need to change the code of the DL training scripts. This middleware must be focused on DL workloads, but without changing the outcome of the *evaluation metric* for any given model. It must allow the efficient data placement of input samples on local disks of compute nodes (*e.g.,* SSD, NVMe) in a transparent manner to the user, while being decloupled from other storage optimizations (*e.g.,* parallel I/O, data preprocessing, sample-level prefetching) that are already implemented in common DL frameworks and libraries.

Chapter 3

PRELIMINARY EXPERIMENTS

To provide accurate predictions, DL models must be trained with large and varied datasets. Moreover, HPC users store these datasets at the infrastructure's PFS. Therefore, storage tiering should be done automatically and transparently for users, in order to leverage the performance benefits of supercomputers local storage resources.

Many studies point to the possibility of using the node's local storage resources as means to speed up DL training. Subsequently, in this work, an experimental evaluation, comparing three different DL training setups that are currently available to users, was conducted.

- ***Lustre***: dataset samples are served from the PFS
- ***Local***: dataset samples are served from the compute node's local storage
- ***Cache***: 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

These setups were specifically chosen to understand and demonstrate the performance impact of running DL jobs under different storage mediums, considering the two extremes scenarios *Lustre* and *Local*, and also the *Cache* intermediate scenario, made possible by a caching mechanism. Furthermore, the following experiment will confirm the properties of the chosen DL models in a given HPC infrastructure, making it known if a model is I/O-bound or compute-bound. It will also serve as a baseline for further comparisons with the solution proposed by this dissertation.

3.1 Experimental Setup

Testbed. Experiments were conducted on a compute node of the Frontera supercomputer [80]. This infrastructure has $448,448$ processing cores and is capable of 39 PFLOPS, being one of the most powerful supercomputers in the world. Table 1 shows the software and hardware for the experiments.

Table 1: Specifications of the experimental environment.

Item	Description
CPU	2x 16-core Intel Xeon E5-2620 v4 (“Broadwell”) Processor
GPU	4x Nvidia Quadro RTX 5000 16GB GDDR6
RAM	128 GiB DDR4
Local storage	119 GiB partition on a 240 GiB SSD
PFS	Lustre
Operating System (OS)	CentOS 7.8
Kernel	version 3.10
File system	XFS
TensorFlow	version 2.3.2
CUDA	version 10.1
Cudnn	version 7.6.5
NCCL	version 2.5.6
Python	version 3.7.0
GCC	version 8.3.0

The compute node memory was limited to 68 GiB to simulate an environment where the entire training dataset would not fit in memory, thus not being fully implicitly cached by the *Page Cache*.

Models. To ensure a comprehensive evaluation, in terms of workload heterogeneity, experiments included three different models, namely ResNet-50, AlexNet and LeNet. These models are based on Concolutional Neural Networks (CNN) architectures [8], and designed to solve image classification problems (*i.e.,* supervised learning). ResNet-50 is the most complex model of the three and is also the one with the most number of layers, requiring a high degree of computational power, as such it is expected to be a compute-bound model. AlexNet has less layers, but more parameters than the ResNet-50. However, AlexNet should represent a slightly less complex model, in comparison to the ResNet-50. Hence, the I/O bottleneck should start to manifest in this model. LeNet is I/O-bound and is the most simple of all three models. Therefore, it is expected that this experiment will have one compute-bound model, and two I/O-bound models.

Dataset. To train the models, a truncated version of the ImageNet-1k dataset [74] was used, that includes $900,000$ images (100 GiB), enabling the dataset to fit entirely on the local storage device. To speedup the training performance, the dataset was converted into the TFRecord optimized data format,

resulting in 1024 TFRecords. The size of each TFRecord varies, since both size *per image* and image quantity *per record* is not constant, however it ranges in average between 90 MiB and 110 MiB.

DL framework. Due to its popularity, the TensorFlow framework was chosen. The model's training input pipeline received a different order of access to the TFRecords (*i.e.,* shuffled file names) for each epoch. Further, the *shuffle stage* (see Section 2.1.9) was also used. These two optimizations are meant to deliver an increased level of randomization for the training process.. Moreover, the constructed model's training input pipeline was also capable of I/O optimizations by using the *interleave stage*, that lead to the parallel loading of files to a buffer, and the *map stage*, which enabled the parallel preprocessing of individual *records*. For the *Cache* setup, the TensorFlow's caching mechanism is enabled with the *cache stage*. The constructed TensorFlow's input pipeline also used the *tf.data.experimental.AUTOTUNE* on the *map* and *interleave stages*, hence the framework defined and optimized the level of parallelism necessary (Section 2.1.9)

Methodology. The elapsed training time and average resource usage (*i.e.,* average CPU, GPU, memory) were measured for all of the conducted experiments. The training scripts were configured to run for 3 training epochs with a 256 batch size [55], and simultaneously use all 4 GPUs available in the compute node, through the *tf.distribute.MirroredStrategy* [58]. The latter functionality divides the batch size across the available GPUs, leaving 64 batched samples for each, in this dissertation's experiments. The results of each experiment concern the average and standard deviation of 7 complete runs. Further, the results displayed in this chapter were derived from the DL framework output (*i.e.,* training time per epoch), dstat [25] (*i.e.,* CPU utilization) and nvidia-smi [62] (*i.e.,* GPU utilization)

3.2 Results

To evaluate the different setups performance, under the previously defined DL models and dataset, it is necessary to examine the training time, and both GPU and CPU utilizations.

Training Time. Figure 13 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 the overall training time for the LeNet and AlexNet models. Under LeNet, the total execution time (*i.e.,* summing the 3 epochs times) decreases from 18.9 to 9.8 minutes (48%), while for AlexNet it decreases from 18.8 to 15.1 minutes (20%). For both models, the decrease in training time is noticeable across all epochs.

With the *Cache* setup, data samples are cached at the compute node's local SSD (*i.e.*, copied from the PFS to the local file system) during the first training epoch. Subsequent epochs fetch data from the

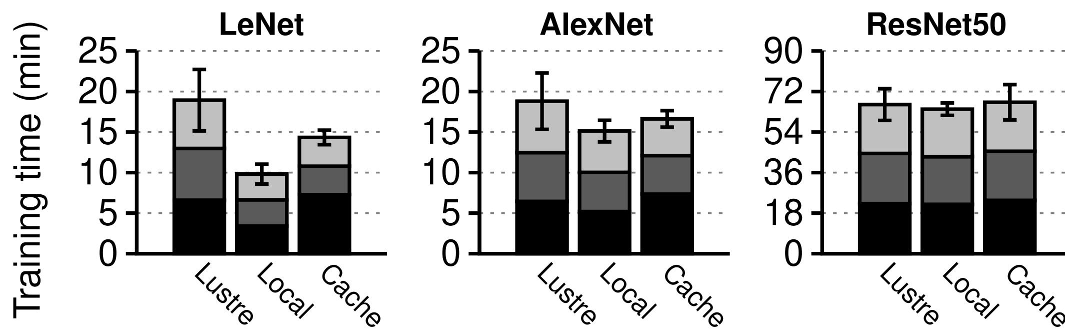


Figure 13: **Average training time** for the *Lustre*, *Local*, and *Cache* setups under LeNet, AlexNet, and ResNet-50 training models. Each column is stacked with the elapsed training time of each training epoch, namely first ( ), second ( ), and third ( ).

local medium, thus reducing the total training time of the LeNet model to 14.4 minutes (24%), and of the AlexNet model to 16.6 minutes (12%), when compared to *Lustre*. The increase in total time, when compared with the *Local* setup is explained by the first epoch of the *Cache* setup, in which there is a slight loss of training performance. In comparison with *Lustre*, the LeNet and AlexNet first epochs show identical results, increasing their times from 6.6 to 7.3 minutes (11%), when using *Cache*. This is explained by the data copying phase of this setup, that must be done between Lustre and the local file system. For the remainder training epochs (*i.e.,* epochs one and two), the training time is very similar to the one achieved by the *Local* setup. For RestNet-50, all setups perform relatively similar, ranging from 64 and 67 minutes of total execution time, hence there are no clear gains, when using local storage mediums, since this model imposes less I/O demand [55].

Interestingly, the *Lustre* setup exhibits the highest training time variability across identical runs of each experiment, which can be observed from the runs' standard deviation. The *Lustre* setup experienced a total of 4 and 3.3 minutes of standard deviation for LeNet and AlexNet, respectively. In contrast and for example, the *Local* setup obtained a total of 1.2 and 2.4 minutes of standard deviation, for the same models, respectively, thus resulting in a 70% and 27% decreases. This is visible for the LeNet and AlexNet 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 unpredictability.

Resource Usage. For I/O-bound models, CPU and GPU usage are related with the throughput at which data samples are fetched from the corresponding storage backends and forwarded to the DL model. If data is ingested at a higher rate, the CPU and GPU will be more occupied in the global time frame, avoiding idle time. It is also important to state that the GPU utilization is proporcional to the model complexity, hence the higher the computational side of a model the longer computations will occur in this device. Thus being said, for both LeNet and AlexNet, 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.

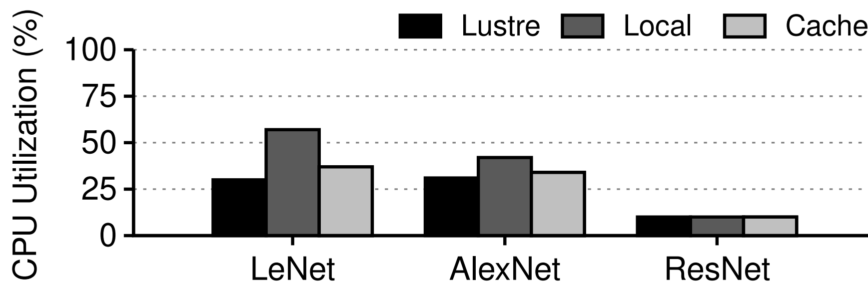


Figure 14: CPU utilization of each model in the different setups.

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

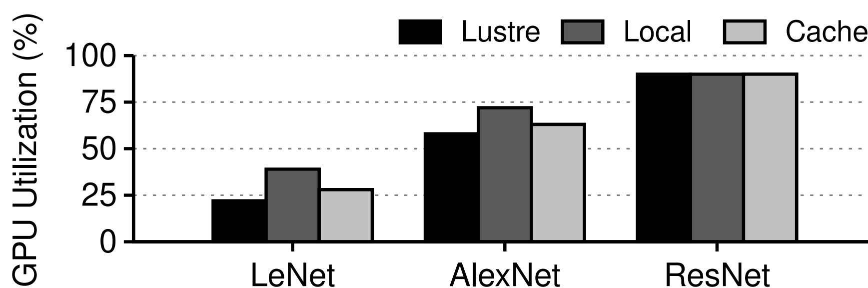


Figure 15: GPU utilization of each model in the different setups.

As expected, the compute-bound ResNet-50 model exhibits the same CPU (10%) and GPU (90%) utilization for all the three setups, having the highest GPU utilization, not necessarily as a result of a high ingestion rate, but due to the complexity of the model and the high number of computations *per batch*. The low CPU usage is explained by the low ingestion and preprocessing rate, that makes the CPU being idle for more periods of the training process. Further, TensorFlow's memory usage is approximately 10 GiB for all models and setups.

3.3 Summary

From this preliminary evaluation, it is confirmed that LeNet and AlexNet remain I/O-bound models for this experimental setup. When the two models are compared, they show very similar results under the *Lustre* setup, however, AlexNet shows less performance improvements in the presence of cached samples, thus it is less I/O dependent and, in fact, more computationally complex than LeNet. In this case, the I/O bottleneck was hiding the complexity of the models, making them perform identically. Moreover, ResNet-50 consistently showed similar results across all setups, hence it's certain that, for the majority of runs, the model will not suffer from the I/O bottleneck, for it is not sufficiently degrading in face of the high amount of computations that must be made in each training step.

To conclude, experiments show that serving the training dataset from local storage backends, which are closer to the computation (*i.e., Local* and *Cache*), can *i)* significantly improve the DL training performance of 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. The *Cache* setup provides transparency, but is limited to scenarios where the full training dataset can fit into the available local storage resources. These limitations are addressed by Monarch, a storage middleware that was developed within the ambit of this dissertation and that will be detailed in the next chapter.

Chapter 4

MONARCH

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

Decoupling. Reading datasets from local storage mediums instead of the PFS proved to be effective to accelerate training times of I/O-bound models [15, 31, 78, 100, 103]. Chapter 3 showed this for TensorFlow with prefetching and parallel I/O enabled. Other DL frameworks with similar access patterns and I/O optimizations should also benefit from storage tiering. Thus, to ensure applicability across different DL frameworks (*e.g.*, TensorFlow, PyTorch) and cross-compatibility with existing I/O optimizations (*e.g.*, data preprocessing, 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.

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 do not fit completely at local storage mediums, Monarch automatically chooses the data samples to keep at each storage tier. Monarch, provides the tools to define a tiering hierarchy, arranging available storage mediums, not requiring further allocation of resource than those that are available at the compute node.

Optimized for DL workloads. Monarch is designed to handle I/O patterns specific to DL training. This dissertation proposes 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 QoS of DL jobs, 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. Thus, Monarch is designed to balance performance gains with the I/O pressure directed at the PFS.

4.1 Architecture

As depicted in Figure 17, 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. This 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*.

4.1.1 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, tiers can be organized in a descending order in terms of performance. An example would be to look at the setup found in Chapter 3, where the local file system could stand as level 1 and the PFS 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.

4.1.2 Placement handler

The *placement handler* is responsible for selecting and fetching dataset files to the correct storage tier and has the following 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 ($[1, N-1]$), leaving the last level untouched (*i.e.,* read-only).

In most DL jobs all dataset files are read at each training epoch, and each file will be read exactly once per epoch. Therefore, Monarch *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 being served by the PFS tier. Since the dataset access may follow a random distribution (*i.e.,* to prevent overfitting [24]), continuously promoting and evicting files (*e.g.,* LRU and Least Frequently Used (LFU)) between storage tiers would increase resource usage and the I/O pressure at the PFS, and would not bring performance improvements to the DL training. Figure 16 is an extension of Figure 8 and shows the benefits of using no eviction under a random access pattern (Section 2.1.7), in detriment to the LRU cache policy. In Figure 16, the cache with LRU obtains 25% and 0% of *cache hit ratio* for the first and second epochs of this example, respectively. However, these values are dependent on the actual access pattern. If in the first epoch after requesting *B* the application, instead of requesting *C*, requested *D*, the value for the *cache hit ratio* of the first epoch would be 50%. On the contrary, since samples are not replaced it is guaranteed that the maximum percentage of *cache hits*, allowed by the cache quota under the DL access pattern in a no-eviction policy, is obtained across all epochs. More specifically, in this illustration, for each epoch and when using a cache with no eviction there is a 50% *cache hit ratio*.

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 it is forwarded immediately to the DL framework. In background, the content of the file is then written (copied) to the appropriate upper tier by following the aforementioned *placement policy*. This asynchronous approach avoids adding extra 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

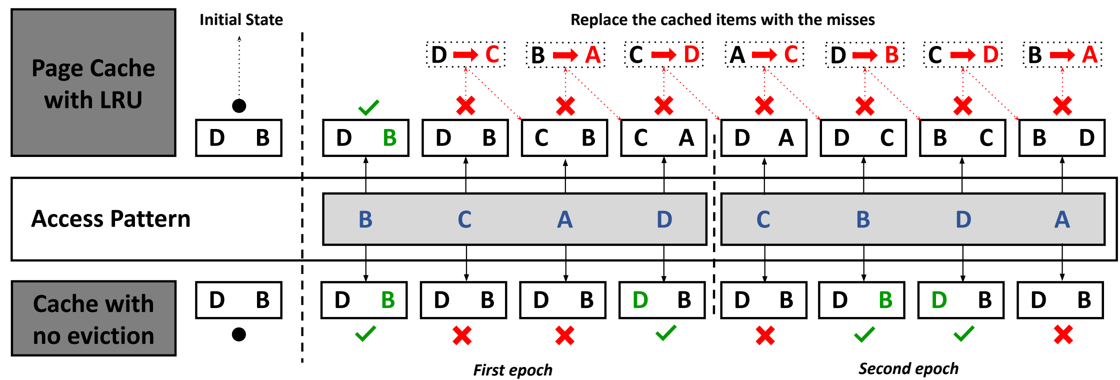


Figure 16: Monarch's cache eviction policy compared with LRU.

content is transparently served to the DL framework from such tier. No additional data placement processing occurs for this case.

Prefetching for Large Files (PLF). 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. When the DL framework is not dealing with large files, but rather small files that can be completely read in one system call, this mechanism does not actuate and the data written to the local storage is the one that was originally read. 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 *evaluation metric*. This aspect, along with the performance benefits of the previous optimizations, are further validated and discussed in Chapter 5.

4.1.3 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 targets DL

frameworks that are using POSIX-compliant backends, it requires two different metadata structures. The first structure allows the mapping of *logical* file paths to *physical* ones, which is important to redirect system calls, such as `open` and `close`. This is the *FPath map* that can be found in Figure 17. The second structure allows the mapping of *logical* file descriptors to *physical* ones to redirect system calls, such as `pread` and `mmap`. This structure corresponds to *FD map* in Figure 17.

4.2 Operation Flow

To understand the mechanisms of Monarch it is essential to describe its general operation flow, which will be based on Figure 17.

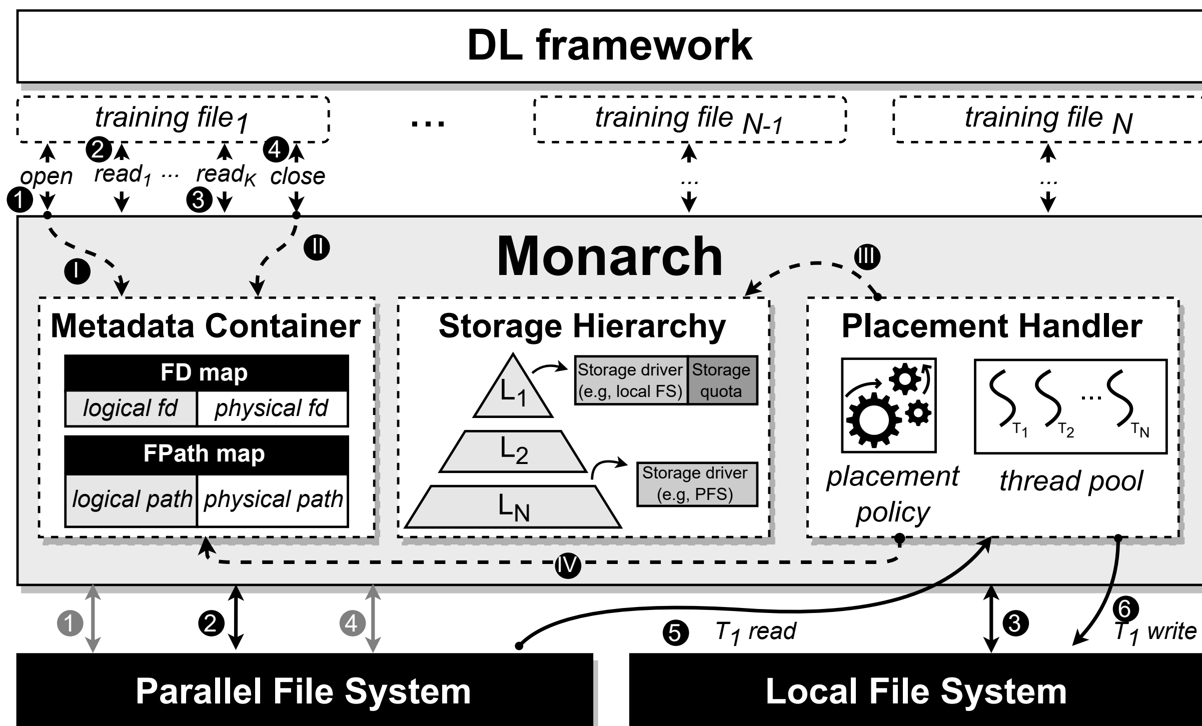


Figure 17: Monarch's architecture and flow of requests.

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 (*e.g.,* Lustre). When the training phase starts, a Monarch instance is initialized, including all of its components. To initialize the *metadata container*, Monarch traverses the directory where the dataset resides (level 2) and builds the necessary metadata information.

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* (Ⅰ) and forwarded to the DL framework. If the file is cached at level 1, which can be consulted at the *metadata container* through the requested file's path, the request is sent to the *file path* at the local file system, and the resulting *fd* is equally stored at the *metadata container* (Ⅰ). 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. Again, for these calls 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* (Ⅱ).

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 (Ⅲ), 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's 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 (Ⅲ). 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. As previously explained, this optimization is applicable for scenarios where the DL framework submits multiple `read` requests to a large file for fetching different data samples.

4.3 Implementation

The Monarch prototype was implemented with 3K lines of C++14. Hence, some crucial implementation details and decisions are further detailed next.

4.3.1 Configuration

Monarch can be easily configured with the help of a *YAML* file. Listing 1 shows how to configure a storage hierarchy with different storage tiers, leaving out other configuration that are less relevant to this explanation, since they are used for debugging, profiling and other experimental features that are under development and are not covered by this dissertation. Thus being said to define a storage tier users must define up to four fields:

- `type`. The kind of the storage backend for that storage tier. Currently, only one type is available, which is the `file_system` type.
- `block_size`. This field defines the block size of the calls issued by Monarch to perform file-based prefetching and can be omitted.
- `max_storage_quota`. Defines the maximum storage quota, in bytes, that can be used to cache data in that tier. This field can be omitted for read-only tiers.
- `mount_point`. Defines the storage tier's mount point directory.

```yaml
# ...                                                          1
# Additional configuration options...                          2
# ...                                                          3
shared_tpool_size: ”6”                                         4
hierarchy:                                                     5
  - type: ”file_system”                                        6
    block_size: ”max”                                          7
    max_storage_quota: ”118111600640”                          8
    mount_point: ”/tmp/monarch_staging_area”                   9
  - type: ”file_system”                                        10
    block_size: ”max”                                          11
    mount_point: ”scratch/user_id/dl_datasets/imagenet-1k”     12
# ...                                                          13
```

Listing 1: Setting up the storage tiers hierarchy with Monarch.

Additionally, the thread pool size can also be specified with the `shared_tpool_size` field. Currently this configuration is passed to Monarch, by defining the environment variable `MONARCH_CONFIGS_PATH`

with the full path to the configuration file. Note that for production this configuration should be defined by the system administrator. A user would then only need to define the full path to its dataset (last storage tier `prefix`), since this varies to all users.

4.3.2 Applicability Across DL Frameworks

Monarch uses the `LD_PRELOAD` technique to alter the linkage of libraries and the resolution of symbols at runtime. This allows Monarch to transparently intercept POSIX calls made by the DL framework, which are destined towards the *logical path*, and route them to the *physical data path* (*i.e.,* appropriate storage tier). Specifically, the `open`, `pread`, `mmap`, and `close` POSIX calls, as displayed Listing 2, supported by `libc`, were replaced by the ones that are serviced by Monarch. Supporting this set of calls is sufficient to attend the requirements of the experiments presented in (Chapter 5).

```
int open (const char* path, int flags, ...);

ssize_t pread (int fd, void *buf, size_t size, off_t offset);

void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);

int close (int fd);
```

Listing 2: Examples of Monarch supported POSIX calls.

4.3.3 Threading and Background Processing

Monarch is designed to work in a single-process multithreaded environment. When a DL framework issues parallel I/O requests, each calling thread will follow an execution path that leads to the execution of Monarch logic, replacing the issued POSIX call. These threads will only have access, as shared state, to the *metadata container* structures to perform Monarch's logic, namely the logic that deals with *I/O calls interception and handling* (Section 4.2).

Monarch utilizes a dedicated thread pool to perform parallel I/O in background, which is triggered after a small read request to a large file is made. When the Prefetching for Large Files (PLF) is needed, the work of this thread pool involves reading files from a storage backend (*e.g.,* PFS) and then writing them to a local storage medium. When the latter is not the case (*i.e.,* the framework's request targets a read of the whole file, usually done when small files are present), then the prefetching mechanism will not be activated. In this case, the file will be completely read by the calling thread. This data will then be used by the thread pool to asynchronously write to the local storage medium. The thread pool is implemented utilizing the C++ Thread Pool Library (CTPL) (version 0.0.2) [20].

4.3.4 Metadata Management

The metadata management is a crucial aspect of Monarch architecture and should be detailed even further, so that its mechanisms are well understood.

To read a file with POSIX, DL frameworks issue multiple system calls. First comes the `open`, then one or many requests to read data (*e.g.,* `pread`, `mmap`), and finally the `close` call. If sequential I/O was enforced, Monarch could easily track consecutive calls and associate them to the desired file, since they always follow the same order, however Monarch deals with multithreaded and interleaved storage I/O, which makes it impossible to correlate consecutive I/O calls that use different arguments to access a file in this manner. Namely, the `open` call uses the file path to access a file and the consecutive calls (*i.e.,* `pread`, `mmap`, `close`) use the returned file descriptor. Therefore, this enforces the need to have the combination of the *FD map* and the *FPath map*, permitting the correct delivery of the desired data.

The *FPath map* is populated during the *metadata container* initialization process and stays as a read-only structure for the entirety of Monarch life cycle. The highly efficient Abseil's (v20210324.2) [3] flat map was used to implement this structure.

The *FD map* starts empty and its state changes during the training phase, when the `open` and `close` calls are issued by the framework, which, respectively, insert and removes entries from the structure. It also enables sharing file descriptors between the DL framework's threads and Monarch's background threads, allowing Monarch's mechanisms to only request the strictly necessary metadata calls. The values inserted into this structure are provided by the *FPath map*. The removal of entries is necessary for the reason that the OS does not perpetually increments/generates new file descriptors for each open, but instead reuses the same ones when a `close` is emitted, thus freeing the file descriptor. Naturally, the updates in this map state may be issued by different DL framework's concurrent threads, hence this structure needs to be thread safe. With this in mind, the *FD map* was built with the Intel Threading Building Blocks Concurrent HashMap (v2021.2.0) [36] library.

Both structures are kept in memory due to performance considerations. Such design does not compromise the fault tolerance of this solution because, if a DL job fails, the *metadata container* information can be initialized again with the data persisted at the PFS.

4.4 Summary

Monarch is a storage middleware solution specifically designed for DL workloads running in HPC systems equipped with multi-tiered storage hierarchies (*e.g.,* shared PFS, node-local SSD). This system targets the performance bottleneck of the DL training phase and aims at decreasing both the training times of I/O-bound models and the pressure exerted to the HPC infrastructure's PFS. Monarch is built on already proven concepts, such as the fact that the node's local storage can be used to partially or completely cache

training dataset, thus potentially accelerating the DL training process, and the use of a non-eviction cache policy to avoid cache *thrashing* in the presence of a random access workload.

Monarch also utilizes I/O-optimizations with the help of background threads to actively and transparently prefetch large files that are being accessed through small reads, which happens for formats such as TFRecords and RecordIO. The proposed solution decouples data staging from the I/O optimizations that are provided by DL frameworks and libraries, and although it offers prefetching, this kind of mechanism is compatible with the prefetching that is already used at the DL framework level, since it is file-based (*i.e.,* prefetching of the whole file) and not sample-based (*i.e.,* prefetching of individual samples or records).

Furthermore, Monarch distances itself from related work solutions that are intrusive to DL developers. It does not require using a specific code library inside of DL training scripts, only requiring the need to specify the location of their training dataset. Moreover, since this system is carefully designed for DL workloads, system administrators do not need to fine-tune any kind of configuration to make it appropriate for this workload, and only need to define the storage tiers hierarchy accordingly.

Monarch has the priority of keeping a balance between performance gains and pressure directed at the PFS, therefore, it has metadata management mechanisms that avoid unnecessary POSIX metadata calls by sharing file descriptors among the involved threads. Moreover, the no-eviction caching policy not only avoids cache *trashing*, but it also prevents the need to be constantly reading the whole dataset, due to the replacement of data files. Some state-of-the-art solutions do not take this into consideration, specially under a single-node training scenario [44, 91]. Finally, Monarch is self-contained, following an ephemeral life cycle that lasts for the duration of the DL job, not needing any kind of external computational resources to hold, for example, metadata.

Chapter 5

EXPERIMENTAL EVALUATION

The evaluation of Monarch seeks to answer the following questions:

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

5.1 Experimental Setup

Testbed, models and datasets. The experimental testbed and models used in these experiments are the same as those described in Chapter 3, however two different datasets, based on the ImageNet-1k, were used for this evaluation process. A *small* version, sizing at 100 GiB, 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 *small* and *large* datasets were converted into TFRecords, resulting in 1024 and 2048 training files, respectively. The doubled number of training files for the large dataset was used to maintain a similar average size for each individual file.

Moreover, the 200 GiB dataset was obtained through data augmentation [67] of images that were present in the original ImageNet-1k dataset and is comprised of roughly 3 million images, which is more than three times the number of images of the 100 GiB dataset. The disproportional growth to the number of images is caused by the augmentation process, which in this case reduced the size of the original

images. This made it so that the dataset needed more images to make it to 200 GiB. This characteristic will serve as a test to evaluate the performance impact of having more records per TFRecord file.

Finally, for experiments where *accuracy* results were needed, a 6.5 GiB *hold-out validation dataset* (Section 2.1.4) with $50,000$ images was used. This dataset was also converted to the TFRecords format, resulting in 128 validation 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.

As a side note, the *metadata container* uses approximately 100 bytes per file entry. Thus for the two datasets not even 1 MiB is used. Further, the initialization process of this module has the duration of 13 seconds for the 100 GiB dataset and 52 seconds for the 200 GiB dataset.

DL frameworks To demonstrate its applicability, Monarch was evaluated under TensorFlow (v2.3.2) and PyTorch (v1.6.0). TensorFlow was set with the same configurations as in Chapter 3. PyTorch experiments were conducted in conjunction with the DALI framework (v.1.5.0) [61].

As mentioned previously in Section 2.2.1, DALI provides better I/O optimizations for PyTorch, replacing the PyTorch dataloader (Section 2.1.10), enabling PyTorch to properly read TFRecords. DALI dataloader needs to be configured with training related parameters, however the I/O optimizations that it provides such as prefetching and parallel I/O are implicit, hence it was only needed to configure the number of working threads.

For the LeNet and AlexNet models 16 threads were used. For the ResNet model this number was reduced to 8, since more threads would cause internal memory allocation errors. Both the DL framework's and the Monarch's number of threads were determined by running preliminary experiments to find a number that reached good performance. This dissertation, however, is not focused on finding the optimal configuration setup to perform DL training, but focused on demonstrating the impact of using Monarch mechanisms on the DL training process. The same can be said about using different DL frameworks. Each DL framework has different data loading mechanisms with different levels of efficiency. It is necessary to study and point out these differences, but it is more important to find how does Monarch impact the DL training performance under different I/O optimizations. Thus, it is not this work's objective to find the optimal DL framework's input pipeline or to strictly compare the different input pipelines.

Methodology The same methodology described in Section 3.1 is applied to this evaluation. In addition, all training experiments, under both PyTorch and TensorFlow DL frameworks, use all 4 GPUs, available at the compute node.

5.2 Results

To answer the previous questions this section will 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 frameworks (TensorFlow and PyTorch) and dataset size (100 GiB and 200 GiB).

5.2.1 TensorFlow 100 GiB

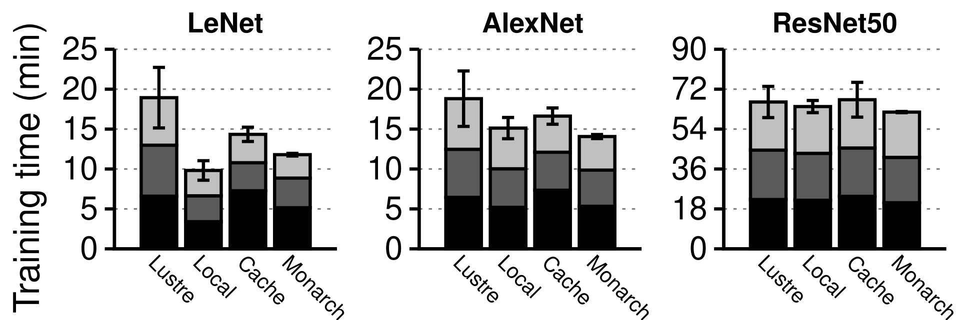


Figure 18: **Average training time** for the *Lustre*, *Local*, and *Cache* setups under LeNet, AlexNet, and ResNet-50 training models for the TensorFlow 100 GiB scenario. Each column is stacked with the elapsed training time of each training epoch, namely first ( ), second ( ), and third ( ).

Training performance. As shown in Figure 18, when compared to *Lustre*, Monarch significantly improves the overall training performance for I/O-bound models, decreasing the training time to 11.8 minutes for LeNet (38% decrease) and to 14 minutes for AlexNet (26% decrease). For ResNet-50, all setups perform similarly.

For the first training epoch (steps $[0, 3500]$), under I/O-bound models, Monarch achieves a better performance than *Lustre* and *Cache*. The LeNet's and AlexNet's first training epoch with Monarch has the duration of roughly 5.3 minutes. For LeNet this poses as a 20% and 27% decreases, in comparison with *Lustre* and *Cache*, respectively. For AlexNet the decrease values are 18% and 28%. Monarch times are similar to those of the *Local* setup, which shows that there is little to no overhead of using this middleware.

The performance gains are explained by Monarch's PLF mechanism. 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 Figure 19, this optimization impacts 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. 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.

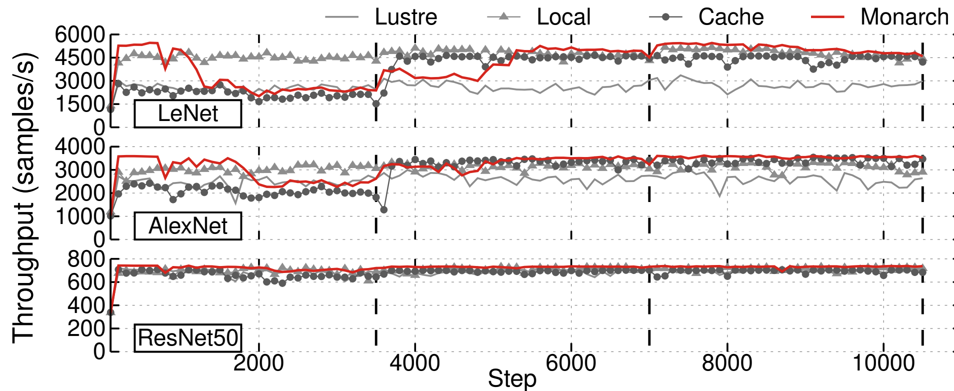
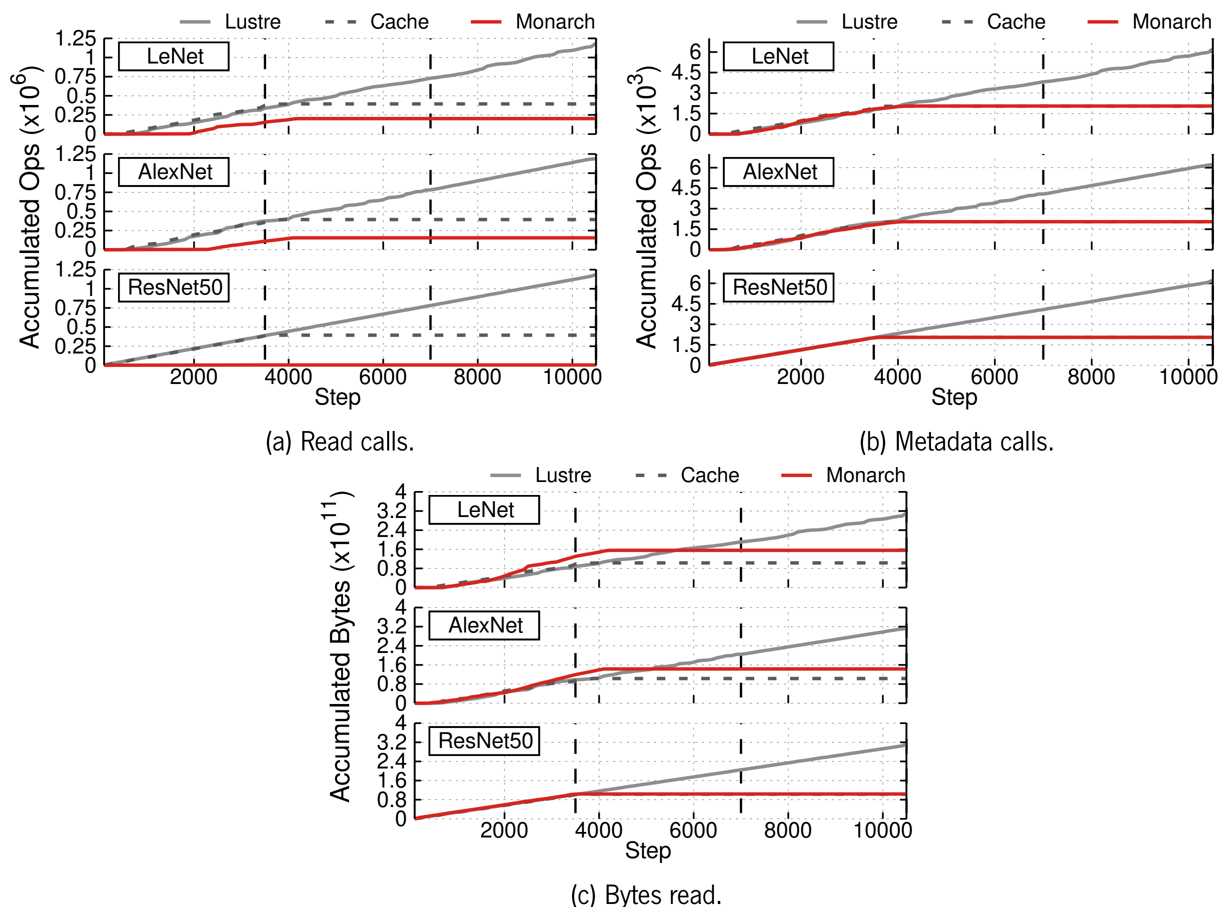


Figure 19: TensorFlow's throughput, in samples *per second*, of *Lustre*, *Local*, *Cache*, and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB.

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 the combined training time for those epochs to 6.7 (46% decrease) and 8.7 (29% decrease) minutes for LeNet and AlexNet, respectively.

PFS operations. As depicted in Figure 20a, due to *storage tiering* and the PLF, Monarch significantly reduces the number of `read` calls directed to the PFS. The *Lustre* setup submits approximately $395,000$ `read` calls per epoch, while *Cache* only submits that same number of `read` calls during the first epoch, since after that, the dataset is served from the local storage tier. The number of reads is not equal to the number of training records (*i.e.,* 900000), since TensorFlow reads chunks of data from the TFRecords (256 KiB each), instead of a single record (Section 2.1.8). These results (*i.e.,* PFS operations) were obtained using the output files of Lustre's `llite`. All of the PFS operations results were obtained using a single separated run (*i.e.,* not using 7 runs to obtain an average and standard deviation), since their variability can be neglected for this evaluation.

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 Monarch's PLF 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, the data staging of files loses performance and cannot keep up with the DL framework rate of requests, thus Monarch submits the small-sized `read` calls to the PFS, while simultaneously storing the dataset in local storage. Finally, when the full dataset is avaliable from the local tier, the placement of files ends and no more `read` calls are submitted to the PFS. Thus, for the LeNet model with Monarch approximately $202,000$ reads were submited to the PFS, while



(a) Read calls.

(b) Metadata calls.

(c) Bytes read.

Figure 20: **TensorFlow 100 GiB PFS operations.** Accumulated read (*a.*), metadata (*b.*) and bytes read (*c.*) operations submitted to the PFS for the *Lustre*, *Cache*, and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for the 100 GiB dataset.

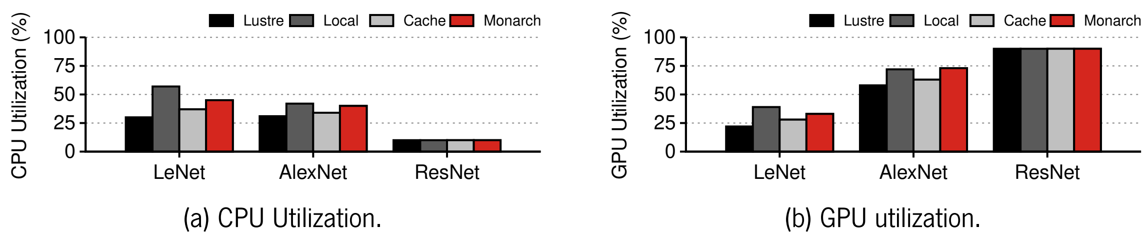
for the AlexNet model the number was $155,000$. With AlexNet, Monarch's PLF is able to prevent more reads requests than for the LeNet.

In addition, Figure 20c shows another perspective of Monarch's PLF mechanism's effects. The loss of data staging performance leads to the small-sized `read` requests being served by the PFS until the desired file is completely prefetched. This means that the DL framework can read a portion of a given file from the PFS and another portion from the local storage, which is backed up by the previous analysis of Figure 20a. Moreover, this will increase the total amount of bytes that are read from the PFS (*i.e.,* originates duplicate data reads). As an example, it is possible, in a worst case and unlikely scenario, that the placement of a file is so delayed that the DL framework finishes reading the whole file without it being placed in the local storage, leading to a double read of the whole file. Therefore, because of Monarch's PLF, for the LeNet model 48 additional GiB were read, whereas for the AlexNet model this value was lower, reaching 36 GiB. This additional amount of read bytes, however proves itself useful, as results show. With it, a decrease in the number of read requests was possible, hence the contention at the PFS was also reduced. Furthermore, this makes a better use of the available bandwidth and reduces each remote I/O request

Round Trip Time (RTT), thus improving training time.

For the ResNet-50 model, since it is compute-bound, Monarch's PLF is able to fetch all data samples timely, only submitting 1024 read calls. In addition, with this model no additional amount of bytes was read from the PFS.

A decrease is also noticeable for metadata operations, namely `open` and `close`. As depicted in Fig. 20b, 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).



(a) CPU Utilization.

(b) GPU utilization.

Figure 21: TensorFlow 100 GiB resource utilization.

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, 40% and 73% for AlexNet, and 10% and 92% for ResNet-50. Regarding memory consumption, Monarch performs identically has the remainder setups (*i.e.,* 10 GiB).

5.2.2 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 Figure 22, Monarch, when compared with *Lustre*, improves the training performance of I/O-bound models, decreasing the training time of the LeNet model from 46.4 minutes to 33.3 minutes (28%), and from 58.3 minutes to 45.9 minutes for the AlexNet model (21%).

Also, one could think that, since the dataset size doubled, the training time should also double, but that did not happen. These results show that, when compared with the previous experiment (*i.e.,* TensorFlow 100 GiB), by having a higher number of records *per TFRecord*, which is the case of the 200 GiB dataset, the training time will increase in an disproportionate manner. This is justified by the extra preprocessing of each

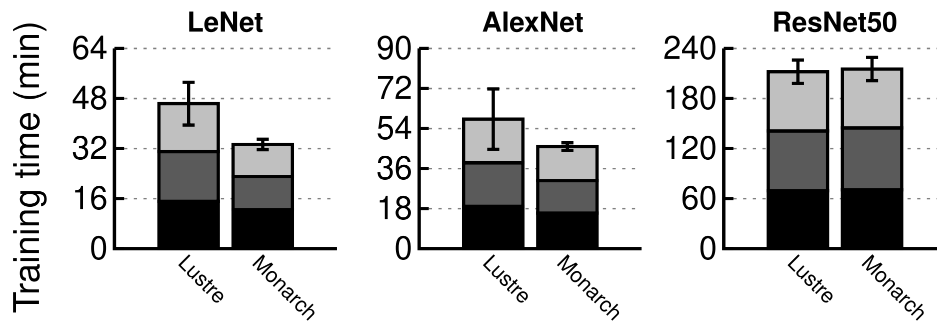


Figure 22: **Average training time** for the *Lustre* and *Monarch* setups under LeNet, AlexNet, and ResNet-50 training models for the TensorFlow 200 GiB scenario. Each column is stacked with the elapsed training time of each training epoch, namely first (▬), second (▬), and third (▬).

record and the additional amount of training *steps*. This is more noticeable in models that involve more computations *per step*. For example, in the 100 GiB experiment, AlexNet and LeNet had similar training times. For this dataset however, that is not observed. Likewise, training ResNet-50 with this dataset lasted three times longer than with the 100 GiB dataset.

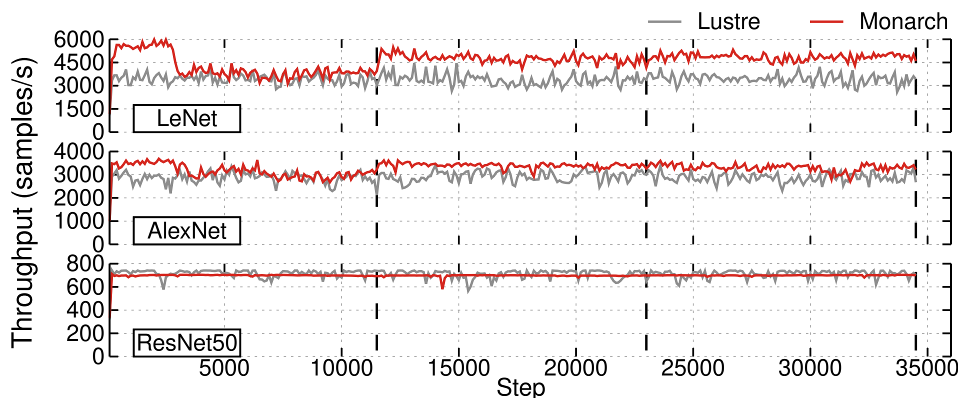


Figure 23: TensorFlow's throughput, in samples *per second*, of *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 200 GiB.

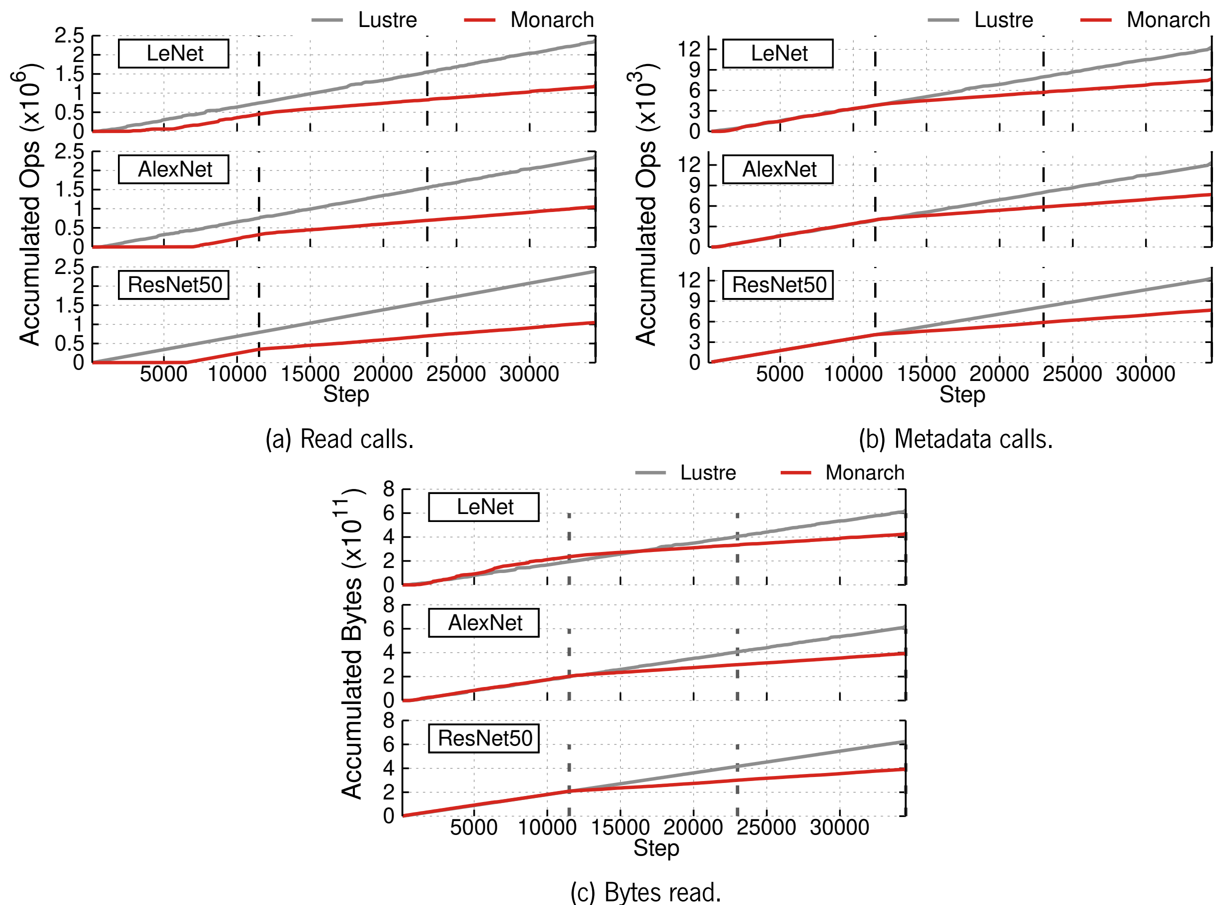
During the first training epoch ($[0, 11500]$), as depicted in Figure 23, Monarch experiences throughput degradation due to the page cache filling up and the local storage tier achieving its quota. For the remainder epochs, Monarch serves the DL framework `read` calls from both local and remote storage tiers. In addition, the dataset's file names are shuffled (*i.e.,* to improve the randomization level) and data samples are being fetched in an interleaved fashion (*i.e.,* using the *interleave stage*), which leads to random throughput variation. With this, throughput spikes can still be seen across different *steps* of the training phase, since the PFS is still being accessed for a large portion of the dataset and in random *steps*. However, the throughput spikes show less amplitude when using Monarch, when compared to *Lustre*.

For the ResNet-50 model, Monarch and *Lustre* perform similarly. In Figure 23 throughput is more stable with Monarch, however this did not have a significant impact on overall the training time.

PFS operations. As depicted in Figure 24a, 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 (*i.e.,* 115 GiB). Specifically, Monarch reduces PFS read operations requests to 1.2 million for the LeNet model (50% reduction) and approximately 1.05 million for the AlexNet and ResNet-50 models (56% reduction, which roughly represents the percentage of the dataset that is cached).

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 Figure 24b.

In terms of bytes read from the PFS, for the same reasons explained in the 100 GiB experiment, for the LeNet model, Monarch read 40 GiB of additional data. For the AlexNet and ResNet-50 models no additional bytes were read from the PFS.



(a) Read calls.

(b) Metadata calls.

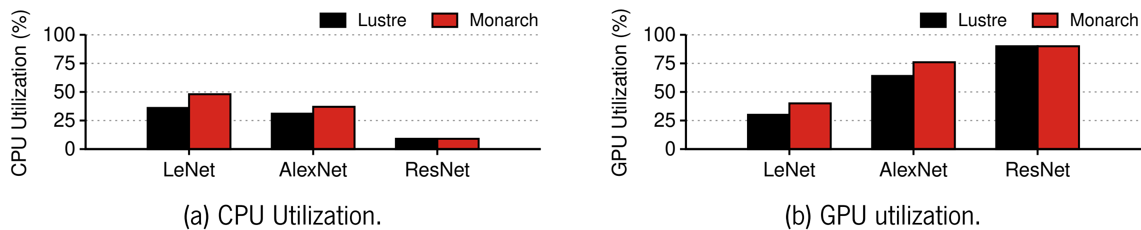
(c) Bytes read.

Figure 24: **TensorFlow 200 GiB PFS operations.** Accumulated read (*a.*), metadata (*b.*) and bytes read (*c.*) operations submitted to the PFS for the *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for the 200 GiB dataset.

Due to the higher number of *records per file* it seems that the AlexNet model with Monarch, unlike the previous 100 GiB scenario, is very similar to the ResNet-50 model in terms of number of operations

submitted to the PFS. This means that the AlexNet model became less I/O-bound and the PLF can now stage TFRecords in a timely fashion, which corroborates what was mention in this experiment training time analysis. Since LeNet has a low number of computations this difference is less noticeable and the staging of files continuous to be unable to keep up with the rate of requests.

Resource usage. Monarch is also 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. In regards to memory consumption, both setups perform similarly (*i.e.,* 10 GiB).



(a) CPU Utilization.

(b) GPU utilization.

Figure 25: TensorFlow 200 GiB resource utilization.

5.2.3 PyTorch 100 GiB

Unlike TensorFlow, PyTorch does not include a persistent caching optimization. Thus, experiments were only conducted over *Lustre*, *Local*, and Monarch setups for the 100 GiB dataset.

Training performance. As depicted in Figure 26, PyTorch exhibits higher training times than TensorFlow for the 100 GiB dataset and all three models, specially for the *Lustre* setup and I/O-bound models.

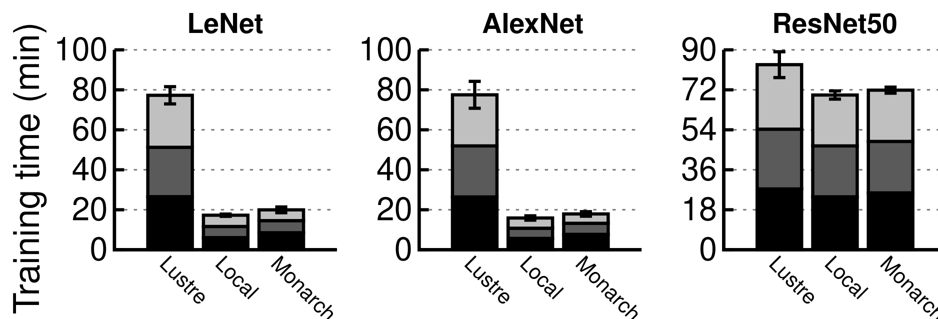


Figure 26: **Average training time** for the *Lustre*, *Local*, and Monarch setups under LeNet, AlexNet, and ResNet-50 training models for the PyTorch 100 GiB scenario. Each column is stacked with the elapsed training time of each training epoch, namely first ( ), second ( ), and third ( ).

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. Again, as with the TensorFlow 100 GiB experiment, the LeNet and AlexNet models showed very close results, however this also stayed true for the *Local* setup, unlike the TensorFlow's experiments. This observation combined with the higher training times for I/O-bound models hints that the data loading optimizations for this experiment are less efficient than those that are used in TensorFlow. One reason for this is the fact that DALI, differently from TensorFlow's pipeline, reads single records from each TFRecord file, instead of reading chunks of data to reduce I/O contention. It is expected that the increase of the number of records (*e.g.,* 200 GiB dataset) will further accentuate this inefficiency.

When compared to *Lustre*, Monarch significantly improves the overall training performance for I/O-bound models, decreasing training time from 77.3 to 20 minutes for LeNet (74%) and from 77.5 to 18 minutes for AlexNet (77%). For ResNet-50, Monarch reduces the training time from 84 to 72.4 minutes (14%). Interestingly, when using PyTorch with DALI, the ResNet-50 model also becomes I/O-bound, thus explaining the performance improvement of Monarch and showing a case for training performance degradation of compute-bound models, in a case where the data loading pipeline does not correctly address the I/O-bottleneck.

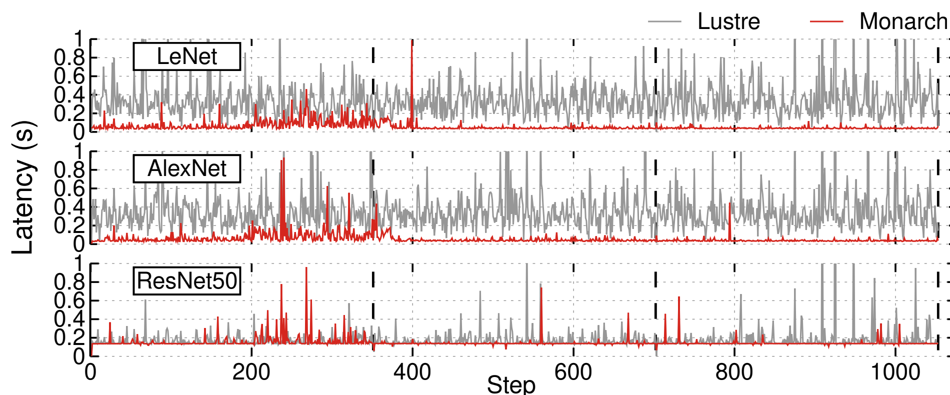
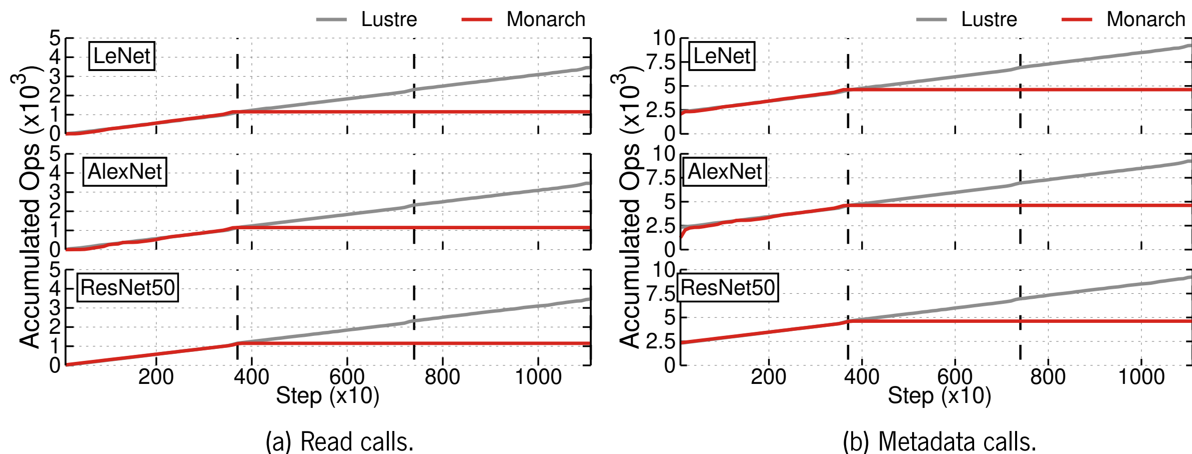


Figure 27: PyTorch's latency, in seconds, of *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 100 GiB.

Local achieves sustained latency per I/O operation throughout the overall execution and across all training models, never exceeding 0.2 seconds. To ease illustration, *Local* results were not included in Figure 27. At the latter figure and for the first training epoch (steps $[0, 3510]$), when compared to *Lustre*, Monarch is able to reduce training time by 68% (18 min difference), 71% (18.7 min difference), and 6% (1.8 min difference) for LeNet, AlexNet, and ResNet-50, respectively. Similarly to the results observed in Section 5.2.1, this is due to Monarch's PLF 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 difference), 80% (40.9 min difference) and 17% (9.7 min difference) for LeNet, AlexNet and ResNet-50, respectively.

PFS operations. The DALI library uses, by default, the `mmap` system call to map whole training files to memory. Contrary to TensorFlow, which performs multiple explicit `read` calls per file. DALI's I/O is performed implicitly when it attempts to access the in-memory data (*i.e.,* as a result of `mmap`) and a *page fault* occurs, resulting in the data samples being copied on demand. As depicted in Figures 28a and 28b, 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. 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. The implicit I/O, however, is not traced by the tools used in this work, which makes it impossible to have plots that illustrate the amount of bytes read, like the TensorFlow experiments, only registering each individual `mmap` call.

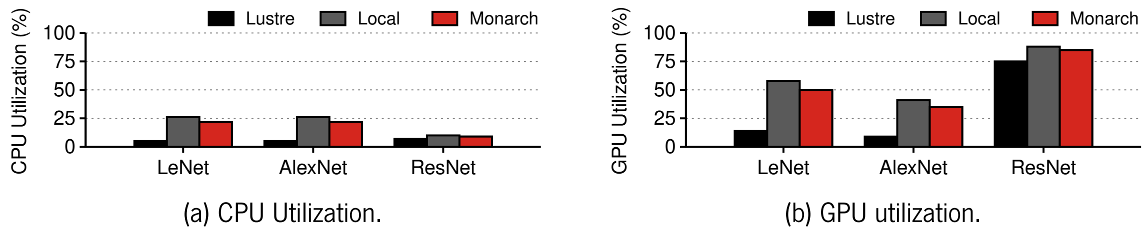


(a) Read calls.

(b) Metadata calls.

Figure 28: **PyTorch 100 GiB PFS operations.** Accumulated read (*a.*) and metadata (*b.*) operations submitted to the PFS for the *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for the 100 GiB dataset.

Resource usage. Monarch shows the second highest CPU and GPU usage, as expected. For LeNet and AlexNet, 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.



(a) CPU Utilization.

(b) GPU utilization.

Figure 29: PyTorch 100 GiB resource utilization.

5.2.4 PyTorch 200 GiB

As in Section 5.2.2, for the 200 GiB dataset, only the *Lustre* and Monarch setups were considered for analysis.

Training time. As depicted in Figure 30, Monarch significantly improves training performance under I/O-bound models, decreasing the training time from 166.4 to 108 minutes (35%) under LeNet and from 165 to 104 minutes (37%). With Monarch the ResNet-50 model training time is reduced by 5% (12 min difference). Again, as discussed in Section 5.2.2, due to the increased number of files for this dataset the training times are much higher.

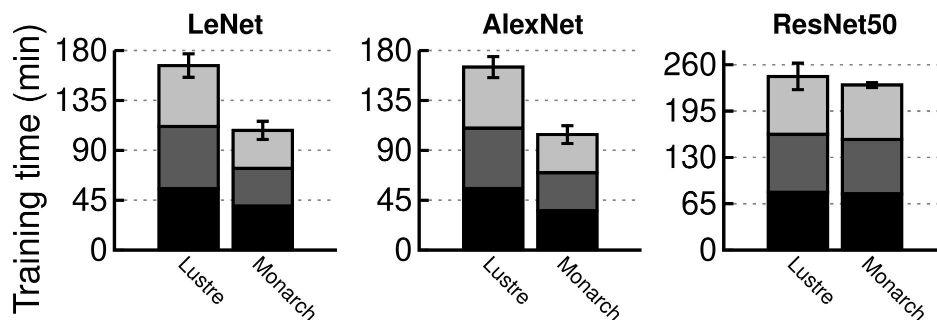


Figure 30: **Average training time** for the *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 training models for the PyTorch 200 GiB scenario. Each column is stacked with the elapsed training time of each training epoch, namely first ( ), second ( ), and third ( ).

Looking at the performance over time (Figure 31), 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 (*i.e.,* 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, very similarly to the TensorFlow's *shuffle buffer*. However the ability to shuffle file names across epochs is not available. 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, and always considering the same order of files).

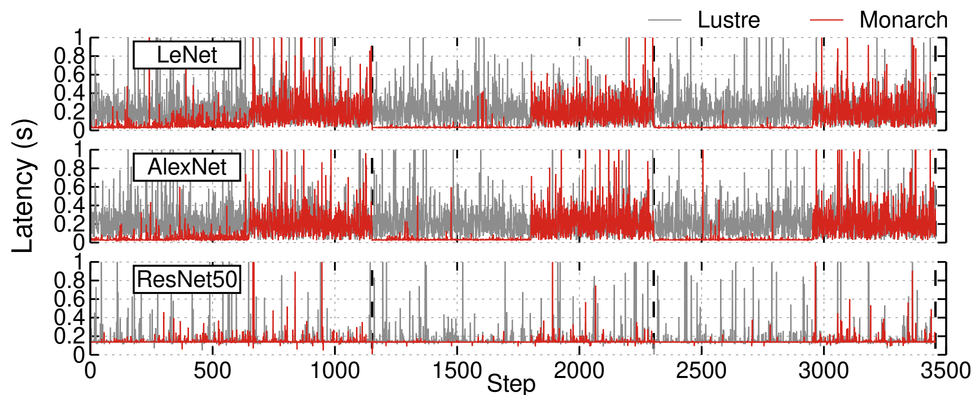
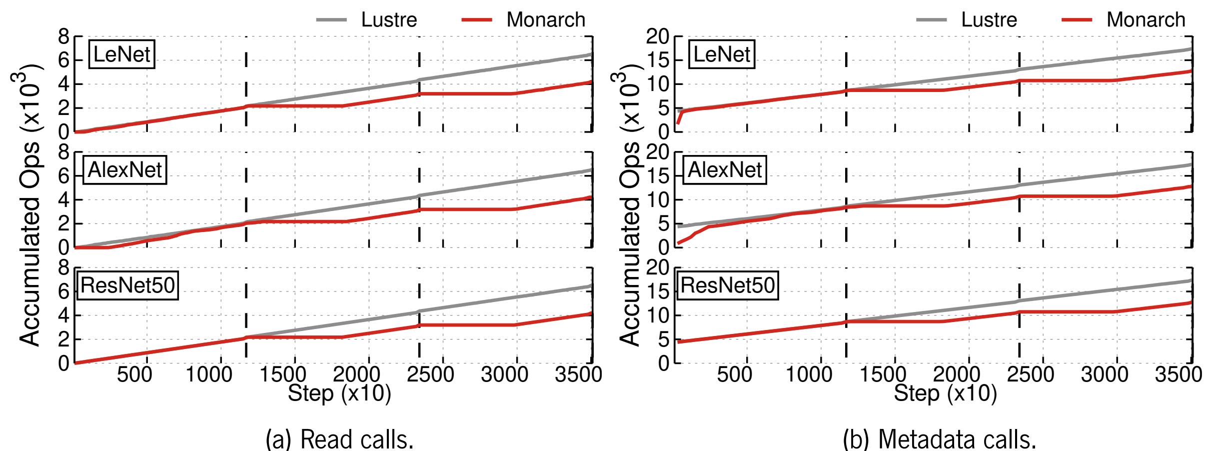


Figure 31: PyTorch's latency, in seconds, of *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for 200 GiB.

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 Section 4.3, 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.



(a) Read calls.

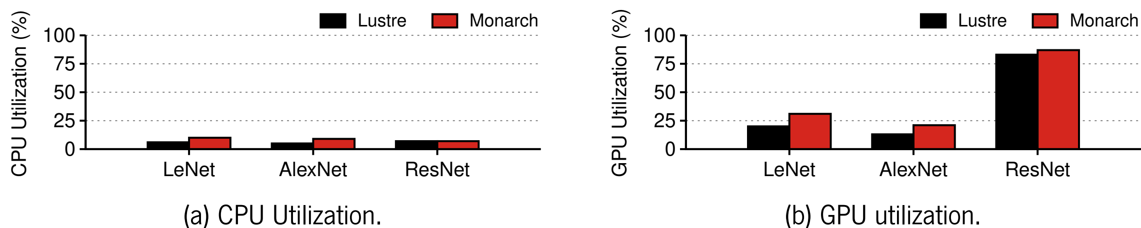
(b) Metadata calls.

Figure 32: **PyTorch 200 GiB PFS operations.** Accumulated read (*a.*) and metadata (*b.*) operations submitted to the PFS for the *Lustre* and Monarch setups under LeNet, AlexNet, and ResNet-50 models, for the 200 GiB dataset.

PFS operations. Similarly to Section 5.2.2, 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.,* roughly 56%). As depicted in Figure 32a, during the first training epoch, both *Lustre* and Monarch perform $1,890$ `mmap` calls. This number is not reduced by Monarch, because, as explained in Section 4.2, what triggers the *prefetching for large files* is, in this case, the `mmap` call, hence all of the calls will be directed at the PFS, only the implicit I/O that comes after the `mmap` is optimized. However, for the remainder epochs, while *Lustre*'s `mmap` calls increase linearly, Monarch submits $2,322$, representing a 50% reduction. This reduction did

not reach the 56% mark, which can be attributed to the order that files were placed. Since some files are larger than others, these will possibly fill up the staging area, leaving more files to be read from the PFS. For the second and third epochs (Figure 32b), 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% (*Lustre*) to 87% (Monarch). Memory consumption results and conclusions are identical to those presented in Section 5.2.3.



(a) CPU Utilization.

(b) GPU utilization.

Figure 33: PyTorch 200 GiB resource utilization.

To conclude, the CPU and GPU utilization values are low when compared to the TensorFlow experiments. The latter, when faced with the 200 GiB dataset, performed similarly to the 100 GiB dataset in terms of resource utilization, however this is not the case for DALI.

5.2.5 Long run and accuracy analysis

Monarch's training performance and *accuracy* were assessed for a 48 hours long training workload (*i.e.,* time limit for regular user's jobs at the Frontera supercomputer). The AlexNet model was chosen, along with the PyTorch deployment and the 200 GiB ImageNet-1k dataset. Next, the results for the *Lustre* and Monarch setups are compared.

As depicted in Figure 34, 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 (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 the previous sections.

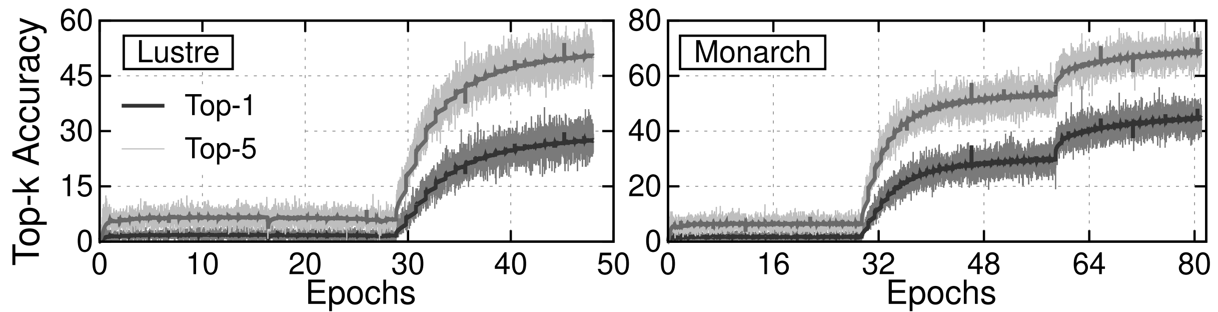


Figure 34: **Top-1 and top-5 *accuracy*** results for PyTorch with *Lustre* and Monarch setups the AlexNet model and 200 GiB dataset, over a 48 hours period.

5.2.6 Discussion

A framework-agnostic middleware like Monarch enables an easy utilization of optimizations, such as data staging and prefetching, which accelerate the training of I/O-bound models.

Notably, besides showcasing Monarch's applicability to different frameworks, models and dataset sizes, the previous experiments validate three key aspects. First, Monarch is able to reduce the training times of I/O-bound models across all experiments, contributing to a substantial mitigation of the I/O bottleneck. In fact, these results correlate into a higher CPU and GPU utilization, which is originated by a higher rate of data ingestion and the minimization of CPU and GPU idle time.

Furthermore, Monarch showed higher performance gains for the first training epoch, when compared with the *Cache* setup, which is justified by the PLF mechanism. With it, Monarch executed larger sequential reads, during placement, taking advantage of the high PFS's aggregate bandwidth. This lead the majority of the DL framework's small read requests to be served from local storage, during the first epoch, thus reducing each request RTT. However, the PLF mechanism was affected by the flushing of the *Page Cache*'s dirty pages to local storage, which made Monarch prefetching mechanism and, consequently, the DL training to lose performance at the time that the *Page Cache* filled up, across all experiments. After the first epoch and for the 100 GiB dataset across different setups, Monarch performed similarly to the *Local* setup, showing that there is no performance overhead, when intercepting POSIX system calls with this storage middleware.

Moreover, for TensorFlow, Monarch achieved training time reduction percentages of up to 38% for the 100 GiB dataset and by up to 28% for the 200 GiB. Naturaly, the acceleration for the 200 GiB dataset is lower than the 100 GiB dataset, since Monarch is only able to partially cache the training dataset. However, this reduction is disproportional. For the 200 GiB dataset, using TensorFlow and the LeNet model, the reduction was not 21% (*i.e.,* the percentage of training time reduction obtained for 56% of dataset caching, considering the reduction value obtained for the 100 GiB). The same happened for the AlexNet model, which went from a 26% reduction (100 GiB dataset) to a 21% reduction (200 GiB dataset), being just a small loss in performance acceleration. These additional performance gains are explained by the higher number of records *per TFRecord file* in the 200 GiB dataset, which is an interesting study case. Therefore,

for the same amount of GiB cached, there are additional training samples placed in the local storage. Since TensorFlow reads the same amount of data in each read request (*i.e.,* each requests reads 256 KiB of data), more *records* are read *per request*, thus Monarch effect on the training acceleration is enhanced.

Likewise, for PyTorch + DALI Monarch reduced the training time by up to 77% for the 100 GiB dataset and 37% for the 200 GiB dataset. Nevertheless, the Pytorch + DALI setup showed a considerably higher I/O-bottleneck than TensorFlow across all models. One of the reasons is that the DALI pipeline reads individual records from each TFRecord file, without requesting chunks of data. This explains the high performance gains with Monarch in the 100 GiB scenario, where hundreds of thousands of requests to read individual records are served from local storage instead of the PFS. For the 200 GiB dataset the performance gains remain disproportional, however and contrary to the TensorFlow case, these gains are smaller than what would be expected (*i.e.,* 43%, which is the percentage of reduction obtained for 56% of dataset caching, having the 100 GiB values as reference). This loss in performance is also attributed to the DALI inefficiency of reading individual records, which imposes a considerable bottleneck for the 200 GiB dataset that cannot be fully solved with caching.

Second, 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%, when using a dataset that does not fit completely on local storage. This is key to ensure stable storage performance for DL workloads and other jobs using the PFS, showing the applicability of Monarch for compute-bond models, where the training performance does not benefit from Monarch mechanisms.

Third and final, Monarch does not impact the *accuracy* of DL workloads, in fact, it enables running more training epochs, and consequently achieving better *accuracy* values, in HPC infrastructures, such as Frontera, were jobs are forced to execute in limited time frames. Moreover, all of this contributions are achieved with a system that has a high level of transparency for users.

Chapter 6

CONCLUSION

The I/O bottleneck is a pressing issue for DL jobs deployed at HPC infrastructures [9, 15, 17, 31, 63]. This problem is substantially exacerbated from the inability to implicitly store the large amounts, both in aggregated size and in number, of training files in memory (*i.e.,* Page Cache or application buffers), therefore the shared PFS takes the toll of having to deal with millions of data and metadata requests per DL job. This not only can cause training performance issues, but it can also lead to the degradation of performance at the PFS and high throughput variability, affecting all PFS users. Although HPC infrastructures' compute nodes often have local storage capabilities that could help to lessen these issues, they are generally overlooked.

With the I/O bottleneck in mind, DL frameworks implement I/O optimizations through custom APIs to be used as iterators and data loaders, however these cannot be shared among different frameworks and have problems of their own. An example of this is TensorFlow's `cache stage`, that does not allow partial data caching. In addition, neither with PyTorch vanilla nor with PyTorch + DALI is a caching mechanism offered.

State-of-the-art solutions that target the performance of scientific workloads consider a wide range of optimization paths, such as deploying or improving the efficiency of data ingestion pipelines, improving I/O parallelism, and even construct I/O buffering systems. These solutions, however, have strong disadvantages, since they can be highly dependent on extensive user configurations or integration on the training scripts code. Some of them are also built for specific DL frameworks. In addition, existing I/O buffering solutions are not correctly optimized for DL workloads and can even change the way data is read, which might lead to unpredictable results to the model's *accuracy*.

Therefore, this dissertation examined, through a preliminary experiment, the effective impact on the training performance of a DL model, when using the compute nodes' local storage to stage training samples. This study showed that I/O-bound models have a clear training acceleration, as well as a decrease in

the performance variation across runs, when using a dataset that is cached at the local storage medium, instead of strictly accessing the PFS.

To address these problems, this dissertation proposes 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, the developed prototype was applied over the 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%.

6.1 Prospects for Future Work

Monarch is a well rounded system built on solid concepts. Nevertheless, there are many investigation paths that can be carried to further enhance its capabilities.

Storage Tiers. Monarch architecture is designed for modular storage tiers. Hence, deeper storage hierarchies, with different tiers (*e.g.,* persistent memory and RAM) can further accelerate the performance of DL training. Moreover, these storage tiers can allow data migration for DL training acceleration. With this, data samples can be transferred between storage tiers, beyond the first training epoch, in case the DL framework's storage access order is known. That order allows data migration from slower storage tiers to faster ones, by identifying what data samples should be transferred, and in what order. The storage access order, however, should only be used in cases where user intervention to determine it is not needed. An example of this is training DL models with the PyTorch + DALI + TFrecords combination, where the dataset's file names cannot be shuffled, thus their access order becomes predictable across epochs.

Autotuning. Monarch has configuration parameters (*e.g.,* `shared_tpool_size`) that can be autotuned to achieve its easier adoption and optimal performance. Autotuning can follow predefined rules that would determine the focus of the tuning. For example, maximizing performance versus minimizing resource usage. Autotuning can also be applied to determine the performance of each storage tier, thus

freeing the HPC infrastructure administrator from the need to specify any meaningful order to their storage tiers. Furthermore, this mechanism can improve the Prefetching for Large Files (PLF) efficiency, by controlling the prefetching and data placement rates (*e.g., proactive draining* [83]) in relation to the DL framework's data ingestion rate, with the goal of minimizing storage tiers' bandwidth and DL frameworks training times. The autotuning approach also lays ways to a more broad solution that considers SDS [56] for infrastructure's holistic autotuning. In this case, Monarch mechanism would be transformed into a *data plane*, having a *control plane*, that has a global view of the HPC center state, to enforce autotuning policies to Monarch instances.

Distributed Training. Applying storage tiering to distributed DL training [10] is challenging. By using multiple compute nodes to train a model the possibility of having distributed memory, as a storage tier arises (*i.e.,* compute nodes can remotely access each others memory). This, however, needs specific placement algorithms that take into consideration which data samples each participant compute node is going to locally cache so that data is shared among compute nodes in the most optimal way, maximizing the use of local storage tiers. This solution requires the need of metadata synchronization and coordination among compute nodes to achieve the best possible results.

Small Files. This dissertation used TFrecords to train DL models, however, serializing small files into a binary file format requires knowledge and a preprocessing step. Therefore, in cases where a datasets with small files are used, it can be effective to implement a transparent small file aggregation/chunking mechanism in the background. Additionally, it is necessary to identify or design an optimized data format for DL workloads that allows efficient data and metadata writes and reads.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. "TensorFlow: A system for large-scale machine learning." In: *12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2016, pp. 265–283.

[2] A. A. Abdulhussein, H. K. Kuba, and A. N. A. Alanssari. "Computer Vision to Improve Security Surveillance through the Identification of Digital Patterns." In: *2020 International Conference on Industrial Engineering, Applications and Manufacturing*. IEEE, 2020, pp. 1–5.

[3] *Abseil*. url: `https://abseil.io/`.

[4] S. Abu-El-Haija, N. Kothari, J. Lee, P. Natsev, G. Toderici, B. Varadarajan, and S. Vijayanarasimhan. *Youtube-8m: A Large-Scale Video Classification Benchmark*. 2016. arXiv: `1609.08675` (cs.CV).

[5] *AI Bridging Cloud Infrastructure*. url: `https://abci.ai`.

[6] A. Aizman, G. Maltby, and T. Breuel. "High Performance I/O For Large Scale Deep Learning." In: *arXiv preprint arXiv:2001.01858* (2020).

[7] T. Alam, S. Qamar, A. Dixit, and M. Benaida. *Genetic Algorithm: Reviews, Implementations, and Applications*. 2020.

[8] S. Albawi, T. A. Mohammed, and S. Al-Zawi. "Understanding of a convolutional neural network." In: *2017 International Conference on Engineering and Technology (ICET)*. IEEE, 2017, pp. 1–6.

[9] *Alibaba Platform for Artificial Intelligence*. `https://www.alibabacloud.com/product/machine-learning`. Accessed December 14, 2021.

[10] T. Ben-Nun and T. Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis." In: *ACM Computing Surveys* 52.4 (2019).

[11] R. Boardman, C. Henninger, and A. Zhu. "Augmented Reality and Virtual Reality: New Drivers for Fashion Retail?" In: *Technology-Driven Sustainability: Innovation in the Fashion Supply Chain*. Springer, 2020, pp. 155–172.

[12] F. A. Breiki, M. Ridzuan, and R. Grandhe. “Self-Supervised Learning for Fine-Grained Image Classification.” In: *CoRR* abs/2107.13973 (2021).

[13] 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. arXiv: 1512.01274 (cs.DC).

[14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. “cuDNN: Efficient Primitives for Deep Learning.” In: *CoRR* (2014). arXiv: 1410.0759.

[15] S. W. D. Chien, S. Markidis, C. P. Sishtla, L. Santos, P. Herman, S. Narasimhamurthy, and E. Laure. “Characterizing Deep-Learning I/O Workloads in TensorFlow.” In: *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems*. IEEE, 2018, pp. 54–63.

[16] D. Choi, A. Passos, C. J. Shallue, and G. E. Dahl. *Faster Neural Network Training with Data Echoing*. 2020. arXiv: 1907.05550 (cs.LG).

[17] 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: *48th International Conference on Parallel Processing*. ACM, 2019.

[18] H. Chu. “MDB: A memory-mapped database and backend for OpenLDAP.” In: *Proceedings of the 3rd International Conference on LDAP, Heidelberg, Germany*. Citeseer. 2011, p. 35.

[19] *CSCS. Piz Daint*. https://www.cscs.ch/computers/piz-daint/. Accessed March 16, 2022.

[20] *CTPL*. url: https://github.com/vit-vit/CTPL.

[21] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. “ImageNet: A large-scale hierarchical image database.” In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2009, pp. 248–255.

[22] J. Devlin, M. Chang, K. Lee, and K. Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL, 2019, pp. 4171–4186.

[23] B. Dong, S. Byna, K. Wu, Prabhat, H. Johansen, J. N. Johnson, and N. Keen. “Data Elevator: Low-Contention Data Movement in Hierarchical Storage System.” In: *2016 IEEE 23rd International Conference on High Performance Computing*. IEEE, 2016, pp. 152–161.

[24] 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*. ACM, 2021.

[25] *dstat(1) - linux man page*. https://linux.die.net/man/1/dstat, note=Accessed February 25, 2022.

[26] *EMC: ABBA*. https://www.theregister.com/2012/09/21/emc_abba/. Accessed January 27, 2022.

[27] C. Francois. *Deep learning with Python*. Manning Publications Company, 2017.

[28] *Frontera Managing I/O Best Practices*. url: https://portal.tacc.utexas.edu/tutorials/managingio#bestpractices-tmp.

[29] *GPUDirect RDMA*. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html. Accessed February 6, 2022.

[30] T. H. Group. *Hierarchical data format version 5*. 2000-2021. url: https://www.hdfgroup.org/solutions/hdf5/.

[31] 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: *2019 IEEE International Conference on Cluster Computing*. IEEE, 2019, pp. 1–12.

[32] *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc, 2019.

[33] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition." In: *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2016, pp. 770–778.

[34] D. Howells and R. Hat. "Fs-cache: A network filesystem caching facility." In: *In Proceedings of the Linux Symposium*. 2006.

[35] *Infinite Memory Engine*. https://www.ddn.com/products/ime-flash-native-data-cache/. Accessed January 27, 2022.

[36] *Intel Threading Building Blocks Concurrent HashMap*. url: https://github.com/oneapi-src/oneTBB.

[37] S. Ioffe and C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: *Proceedings of the 32nd International Conference on International Conference on Machine Learning*. JMLR.org, 2015, pp. 448–456.

[38] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. "Caffe: Convolutional Architecture for Fast Feature Embedding." In: *Proceedings of the 22nd ACM International Conference on Multimedia*. ACM, 2014, pp. 675–678.

[39] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit." In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 2017, pp. 1–12.

[40] J. M. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Zídek, A. Potapenko, A. Bridgland, C. Meyer, S. A. A. Kohl, A. Ballard, A. Cowie, B. Romera-Paredes, S. Nikolov, R. Jain, J. Adler, T. Back, S. Petersen, D. A. Reiman, E. Clancy, M. Zielinski, M. Steinegger, M. Pacholska, T. Berghammer, S. Bodenstein, D. Silver, O. Vinyals, A. W. Senior, K. Kavukcuoglu, P. Kohli, and D. Hassabis. "Highly accurate protein structure prediction with AlphaFold." In: *Nature* (2021), pp. 583 –589.

[41] *Keras*. https://keras.io/. Accessed December 16, 2021.

[42] M. Khanum, T. Mahboob, W. Imtiaz, H. Ghafoor, and R. Sehar. "A Survey on Unsupervised Machine Learning Algorithms for Automation, Classification and Maintenance." In: *IJCA* (2015), pp. 34–39.

[43] S. B. Kotsiantis. "Supervised Machine Learning: A Review of Classification Techniques." In: *Proceedings of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in EHealth, HCI, Information Retrieval and Pervasive Technologies*. IOS Press, 2007, pp. 3–24.

[44] A. Kougkas, H. Devarajan, and X.-H. Sun. "Hermes: A Heterogeneous-Aware Multi-Tiered Distributed I/O Buffering System." In: *27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2018, pp. 219–230.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." In: (2017), pp. 84–90.

[46] A. V. Kumar and M. Sivathanu. "Quiver: An Informed Storage Cache for Deep Learning." In: *18th USENIX Conference on File and Storage Technologies*. USENIX, 2020, pp. 283–296.

[47] T. Kurth, S. Treichler, J. Romero, M. Mudigonda, N. Luehr, E. Phillips, A. Mahesh, M. Matheson, J. Deslippe, M. Fatica, Prabhat, and M. Houston. "Exascale Deep Learning for Climate Analytics."

In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE, 2018.

[48] F. P. Lanaras. "Reducing data path from storage to GPUs for Deep Learning." In: 2018.

[49] B. R. Landsteiner, D. Henseler, D. Petesch, and N. J. Wright. "Architecture and Design of Cray DataWarp." In: *Cray User Group Conference*. 2016.

[50] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, et al. "Gradient-based Learning Applied to Document Recognition." In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[51] S.-H. Lim, S. Young, and R. Patton. "An analysis of image storage systems for scalable training of deep neural networks." In: Apr. 2016.

[52] N. Liu, J. Cope, P. Carns, C. Carothers, R. Ross, G. Grider, A. Crume, and C. Maltzahn. "On the role of burst buffers in leadership-class storage systems." In: *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies*. IEEE, 2012, pp. 1–11.

[53] 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, pp. 931–943.

[54] 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: *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2010, pp. 1–12.

[55] 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: *2021 IEEE International Conference on Cluster Computing*. IEEE, 2021, pp. 649–656.

[56] R. Macedo, J. Paulo, J. Pereira, and A. Bessani. "A Survey and Classification of Software-Defined Storage Systems." In: *ACM Computing Surveys* 53.3 (2020).

[57] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu. "Convergence analysis of distributed stochastic gradient descent with shuffling." In: *Neurocomputing* 337 (2019), pp. 46–57.

[58] *Mirrored Strategy*. https://www.tensorflow.org/api_docs/python/tf/distribute/MirroredStrategy. Accessed February 14, 2022.

[59] J. Mohan, A. Phanishayee, A. Raniwala, and V. Chidambaram. "Analyzing and Mitigating Data Stalls in DNN Training." In: *Proceedings of the VLDB Endowment* 14.5 (2021), pp. 771––784.

[60] A. Y. Ng. "Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance." In: *Proceedings of the Twenty-First International Conference on Machine Learning*. ACM, 2004.

[61] *NVIDIA Data Loading Library*. url: https://developer.nvidia.com/dali.

[62] *Nvidia system management interface*. https://developer.nvidia.com/nvidia-system-management-interface. Accessed February 25, 2022.

[63] L. Oden, C. Schiffer, H. Spitzer, T. Dickscheid, and D. Pleiter. "IO Challenges for Human Brain Atlasing Using Deep Learning Methods - An In-Depth Analysis." In: *27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. 2019, pp. 291–298.

[64] H. Ohtsuji, E. Hayashi, and N. Fukumoto. "Mitigating the Impact of Tail Latency of Storage Systems on Scalable Deep Learning Applications." In: *Parallel Data Systems Workshop*. 2019.

[65] *Open Images Dataset*. 2017. url: https://github.com/cvdfoundation/open-images-dataset.

[66] 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." In: *NIPS 2017 Workshop Autodiff*. 2017.

[67] L. Perez and J. Wang. *The Effectiveness of Data Augmentation in Image Classification using Deep Learning*. 2017. arXiv: 1712.04621 [cs.CV].

[68] *Pinned Memory*. https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/. Accessed January 20, 2021.

[69] S. Pumma, M. Si, W.-C. Feng, and P. Balaji. "Scalable Deep Learning via I/O Analysis and Optimization." In: *ACM Transactions on Parallel Computing* 1.1 (2019), pp. 1–34.

[70] *Pytorch Dataloader*. https://pytorch.org/docs/stable/data.html. Accessed November 12, 2021.

[71] Y. Qian, X. Li, S. Ihara, A. Dilger, C. Thomaz, S. Wang, W. Cheng, C. Li, L. Zeng, F. Wang, D. Feng, T. Süß, and A. Brinkmann. "LPCC: Hierarchical Persistent Client Caching for Lustre." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2019.

[72] *RecordIO*. https://mxnet.apache.org/versions/1.8.0/api/python/docs/api/mxnet/recordio/index.html. Accessed December 15, 2021.

[73] *RIKEN Center for Computational Science, About Fugaku*. https://www.r-ccs.riken.jp/en/fugaku/about/. Accessed March 16, 2022.

[74] 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." In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

[75] *Scaling Uber's Apache Hadoop Distributed Filesystem for Growth*. https://eng.uber.com/scaling-hdfs/. Accessed January 31, 2021.

[76] F. B. Schmuck and R. L. Haskin. "GPFS: A Shared-Disk File System for Large Computing Clusters." In: *1st USENIX Conference on File and Storage Technologies*. USENIX, 2002, pp. 231–244.

[77] P. Schwan. "Lustre: Building a File System for 1000-node Clusters." In: *Proceedings of the 2003 Linux Symposium*. 2003, pp. 380–386.

[78] K. Serizawa and O. Tatebe. "Accelerating Machine Learning I/O by Overlapping Data Staging and Mini-Batch Generations." In: *6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*. 2019, pp. 31–34.

[79] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." In: *Journal of Machine Learning Research* (2014), pp. 1929–1958.

[80] 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*. ACM, 2020, pp. 106–111.

[81] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[82] V. Sze, Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey." In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[83] K. Tang, P. Huang, X. He, T. Lu, S. S. Vazhkudai, and D. Tiwari. "Toward Managing HPC Burst Buffers Effectively: Draining Strategy to Regulate Bursty I/O Behavior." In: *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2017, pp. 87–98.

[84] *TensorFlow API: tf.data.Dataset.cache*. url: https://www.tensorflow.org/api_docs/python/tf/data/Dataset\#cache.

[85] *Tensorflow Input Pipeline*. https://www.tensorflow.org/guide/data. Accessed November 12, 2021.

[86] *TFRecord*. url: https://www.tensorflow.org/tutorials/load_data/tfrecord.

[87] S. Tokui, K. Oono, S. Hido, and J. Clayton. "Chainer: a next-generation open source framework for deep learning." In: *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems*. Vol. 5. 2015, pp. 1–6.

[88] *Understanding the Linux Virtual Memory Manager*. https://www.kernel.org/doc/gorman/html/understand/understand013.html. Accessed December 15, 2021. 2020.

[89] N. Vasilache, J. Johnson, M. Mathieu, S. Chintala, S. Piantino, and Y. LeCun. "Fast Convolutional Nets With fbfft: A GPU Performance Evaluation." In: *arXiv preprint arXiv:1412.7580* (2015).

[90] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann. "GekkoFS - A Temporary Distributed File System for HPC Applications." In: *2018 IEEE International Conference on Cluster Computing*. 2018, pp. 319–324.

[91] 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: *49th International Conference on Parallel Processing*. ACM, 2020.

[92] M. Wang, C. Meng, G. Long, C. Wu, J. Yang, W. Lin, and Y. Jia. "Characterizing Deep Learning Training Workloads on Alibaba-PAI." In: *2019 IEEE International Symposium on Workload Characterization*. 2019, pp. 189–202.

[93] T. Wang, K. Mohror, A. Moody, K. Sato, and W. Yu. "An Ephemeral Burst-Buffer File System for Scientific Applications." In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 807–818.

[94] C. Yang and G. Cong. "Accelerating Data Loading in Deep Neural Network Training." In: *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics*. 2019, pp. 235–245.

[95] Y. Yao, L. Rosasco, and A. Caponnetto. "On Early Stopping in Gradient Descent Learning." In: *Constructive Approximation* 26 (2007), pp. 289–315.

[96] 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: *2016 IEEE International Parallel and Distributed Processing Symposium*. IEEE. 2016, pp. 750–759.

[97] B. Zamanlooy and M. Mirhassani. "Efficient VLSI Implementation of Neural Networks With Hyperbolic Tangent Activation Function." In: *IEEE Transactions on Very Large Scale Integration Systems* 22 (2014), pp. 39–48.

[98] M. Zeiler, M. Ranzato, R. Monga, M. Mao, K. Yang, Q. Le, P. Nguyen, A. Senior, V. Vanhoucke, J. Dean, and G. Hinton. "On rectified linear units for speech processing." In: *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. 2013, pp. 3517–3521.

[99] D. Zhang, S. Mishra, E. Brynjolfsson, J. Etchemendy, D. Ganguli, B. Grosz, T. Lyons, J. Manyika, J. C. Niebles, M. Sellitto, Y. Shoham, J. Clark, and R. Perrault. "The AI Index 2021 Annual Report." In: *AI Index Steering Committee, Human-Centered AI Initiative, Stanford University* (2021).

[100] 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. arXiv: 1809.10799 (cs.DC).

[101] Q. Zheng, C. D. Cranor, G. R. Ganger, G. A. Gibson, G. Amvrosiadis, B. W. Settlemyer, and G. A. Grider. "DeltaFS: A Scalable No-Ground-Truth Filesystem for Massively-Parallel Computing." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2021.

[102] 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: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2018, pp. 145–156.

[103] Y. Zhu, W. Yu, B. Jiao, K. Mohror, A. Moody, and F. Chowdhury. "Efficient User-Level Storage Disaggregation for Deep Learning." In: *2019 IEEE International Conference on Cluster Computing*. 2019.