ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Clairvoyant Prefetching for Machine Learning I/O

Bachelor Thesis

Roman Böhringer

July 23, 2020

Supervisor: Prof. Dr. Torsten Hoefler
Advisors: Dr. Tal Ben-Nun, Dr. Nikoli Dryden

Department of Computer Science, ETH Zürich

Abstract

There has been significant interest in deep neural networks recently, both from a theoretical and practical perspective. Many researchers have proposed new schemes and methods to minimize the training time of the networks, especially on high performance computing clusters. While we have seen tremendous progress in the training times, data loading is often neglected in these works and is becoming a major bottleneck for training. We expect this trend to continue, as datasets are becoming larger and specialized hardware for accelerating machine learning is more widely deployed. In this work, we study the data loading problem from a theoretical perspective and use these insights to design and implement a novel machine learning I/O middleware, HDMLP. The framework provides an easy-to-use, flexible solution that scales well to many nodes and large datasets, which makes it especially suitable for distributed deep learning on supercomputers. We confirm in theoretical and practical evaluations that the solution delivers better performance than current state-of-the-art approaches (reducing the median stall time by factors of up to 44) while requiring very few changes to existing codebases and supporting a broad range of environments.

Acknowledgements

I want to express my deepest gratitude to Nikoli Dryden and Tal Ben-Nun for their valuable inputs during my work on the thesis. My sincere thanks also goes to Prof. Torsten Hoefler for allowing me to work on this thesis in the scalable parallel computing laboratory and granting me access to all the resources I required.

Contents

List of Figures

List of Tables

List of Source Code Listings

Chapter 1

Introduction

Machine learning and especially deep learning is increasingly used in industry and research. We have seen successful implementations of these methods in areas such as medicine [42], autonomous driving [18], climate analytics [28], cosmology [32], and many more. Besides an increase in DNN complexity, the datasets that are used have grown in size, with some of them reaching sizes of up to several petabytes [15]. Specialized hardware accelerators for machine learning are becoming more common [24,31] and there has been much research on optimizing the performance of neural network training, especially on high-performance computing clusters [17,22,35,43,45,46]. Most of these works optimized hyperparameters or algorithms of the training process and neglected data loading, which is becoming a major bottleneck, especially for distributed training, as the bandwidth of the parallel file system can be saturated quickly when many nodes are used for training. The file access pattern of machine learning training appears arbitrary to the file system because of stochastic gradient descent's random nature. Therefore, many traditional I/O optimizations that take advantage of spatial or temporal locality do not perform well. However, we can make use of the fact that the access pattern is inferable by the machine learning framework before training even starts as it is completely determined by the state of the pseudorandom number generator.

Besides performance optimizations, a lot of machine learning frameworks such as PyTorch [36], TensorFlow [1], Caffe [23], MXNet [11], and others have been developed, which considerably simplify the implementation of new machine learning models. To our knowledge, there currently exists no general machine learning I/O framework which similarly simplifies data loading while also providing performance benefits, scalability, and other desired properties.

In this work, we analyze the existing solutions that are used to optimize I/O performance for deep learning with a focus on distributed deep learning.

We define properties that an ideal solution should fulfill (dataset & node scalability, configuration independence, model accuracy, and ease of use) and analyze the solutions with regard to these characteristics. We then continue to develop an analytical performance model for the problem which we use to compare existing solutions. Based on the performance model and other problem-specific insights (among others that using the access frequency for the prefetching decision results in a good sample distribution while also reducing the stall times of individual nodes), we design a novel data loading, caching, and prefetching system for machine learning I/O that fulfills all criteria. We implement the framework in C++/Python and describe its implementation in detail. Finally, we analyze the performance of our solution, both theoretically and with real benchmarks on a high-performance cluster. In our benchmarks, we were able to reduce the median stall time by factors of up to 44 compared to PyTorch's multi-threaded prefetcher with minimal code changes by the end user.

To summarize, the main contributions of this work are:

- We survey current solutions for data loading, caching, and prefetching in the context of distributed machine learning.
- We develop an analytical performance model for modeling the runtime of distributed machine learning. Our model especially highlights the impact of data loading on the execution time.
- We analyze and compare existing solutions with our performance model.
- We use the performance model and other problem-specific insights to design and implement HDMLP, the Hierarchical Distributed Machine Learning Prefetcher. The solution is a general machine learning I/O framework/middleware that is easy to deploy, results in good performance, can be used in very different environments, and scales well to many nodes and large datasets.

Chapter 2

Background

2.1 Machine Learning

Machine learning refers to the automated extraction of meaningful patterns in data and has become very common for tasks where information extraction from large datasets is required [38]. In contrast to traditional algorithms, a programmer does not specify an explicit recipe for the solution of a problem, but an algorithm to infer these solutions instead. One type of machine learning is deep learning. The name artificial neural networks is often used when referring to deep learning because some of its earliest algorithms were intended to be models of human learning in the brain [29]. Although deep learning has been around for a long time, it started to gain a lot of attention in the 2010s when deep neural networks (DNNs) outperformed systems based on other machine learning techniques and hand-designed systems [16]. Two major factors in this development (that continues to this day) were the increasing dataset sizes and the larger model sizes which were made possible by technological advancements and more computational power [16].

In deep learning we try to approximate some function f^*, for instance a classifier $y = f^*(\mathbf{x})$ that maps an input $\mathbf{x}$ to a category y. We define a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ and want to learn the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation of f^*, where the quality is measured with respect to some loss function L (a function representing the difference between the desired output and the actual output). The networks are typically represented by composing together many different functions and the model is associated with a directed acyclic graph that defines the function composition. The dominant training algorithm for deep learning models today is stochastic gradient descent (SGD) [16]. To minimize the expected loss on the training set, dataset elements are sampled at random, either in mini-batches of size m or one-at-a-time (corresponding to $m = 1$). The gradient estimate is computed as the average of the m gradients, after which the parameter update rule is applied.

Algorithm 1: Stochastic Gradient Descent [16].

```
Require: Learning rate schedule $\varepsilon_1, \varepsilon_2, \ldots$
Require: Initial parameter $\boldsymbol{\theta}$
$k \leftarrow 1$;
while stopping criterion not met do
    Sample mini-batch of $m$ examples from the training set
      $\{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ with corresponding targets $\mathbf{y}^{(i)}$;
    Compute gradient estimate: $\hat{\mathbf{g}} \leftarrow \frac{1}{m}\nabla_\theta \sum_i L(f(\mathbf{x}^{(i)}; \boldsymbol{\theta}), \mathbf{y}^{(i)})$ ;
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon_k \hat{\mathbf{g}}$ ;
    $k \leftarrow k + 1$ ;
end
```

To compute an unbiased estimate of the expected gradient, the samples need to be independent which implies that the mini-batches are selected uniformly at random [16] (typically without replacement [39]).

2.1.1 Distributed Machine Learning

There are three major strategies for partitioning the training of a deep neural network among multiple devices/nodes and several hybrid schemes that combine these approaches [6]:

1. **Data Parallelism**: In this approach, the mini batches of size m are further partitioned into worker-local batches that are processed by individual workers (where a worker can for instance be a node, processor, or GPU, depending on the setup). The weight updates produced by each partition are averaged, which is generally done with an allreduce operation. The method requires all DNN parameters to be replicated among all nodes.

2. **Model Parallelism**: In model parallelism (also known as network parallelism), different parts of the DNN are computed on different workers by distributing the work per layer (a collection of nodes/functions that operate together at a specific neural network depth) among different workers. Therefore, not all workers require all DNN parameters which can reduce the memory usage. However, the communication overhead is generally higher as the strategy requires additional communication after every layer (e.g. all-to-all communication for fully-

connected layers, whereas data parallelism only performs an allreduce operation).

3. **Pipelining**: Pipelining in the context of deep learning either means overlapping computations (feeding the results of one layer to the next one as data becomes ready) or assigning different layers of the neural network to different workers. In the latter approach, not all parameters need to be stored on every worker and communication is only performed at the layer boundaries.

2.2 Supercomputer Storage Configurations

There is a wide variety of storage configurations among different supercomputers. Some systems like Piz Daint[1] at the Swiss National Supercomputing Centre or SuperMUC-NG[2] at Leibniz Supercomputing Centre are equipped with no node-local storage. On the other hand, there are systems like Sierra[3] at Lawrence Livermore National Laboratory with a 1.6TB NVMe PCIe SSD per compute node or Summit[4] at Oak Ridge National Laboratory where every node has additional 800GB of non-volatile memory. Besides these on-node burst buffers, there are also systems with dedicated burst buffer nodes that are located alongside I/O nodes and systems with global burst buffers [25].

We therefore should not assume anything about the storage configuration of the target system because there are great differences. The only thing that all systems in high-performance computing environments have in common is that they can read from and write to a parallel file system (PFS) that is accessible by all nodes and usually slower than local storage.

Parallel file systems are typically designed and optimized for large and sequential I/O, which is a common access pattern for scientific applications. The access pattern of SGD with random accesses to samples that are often relatively small imposes significant pressure on the file system, especially when the number of nodes is increased [13]. Furthermore, there are environments where the network bandwidth for communication among nodes is much higher than the network bandwidth for communication with the PFS, which can also become a bottleneck when scaling up the number of nodes.

[1] https://www.cscs.ch/computers/piz-daint/
[2] https://doku.lrz.de/display/PUBLIC/SuperMUC-NG
[3] https://hpc.llnl.gov/hardware/platforms/sierra
[4] https://www.olcf.ornl.gov/summit/

Approach	Dataset Scalability	Node Scalability	Configuration Independence	Model Accuracy	Ease of Use
tf.data with `shuffle/prefetch` [12]	✓	✗	✗	✗	✓
Double Buffering	✓	✗	✗	✓	✓
DeepIO [47]	✓	✓	✗	✗[a]	✓
Parallel Data Staging [28]	✓	✓	✗	✗	✗
LBANN Distributed Data Store [21]	✗	✓	✗	✓	✗
Locality-Aware Data Loading [44]	✓	✓	✗	✓[b]	?[c]

[a]In entropy-aware opportunistic mode
[b]Result can be different when using node-local batch normalization
[c]PyTorch based prototype, general software package is planned according to the paper

Table 2.1: An Overview of Existing Solutions.

2.3 Challenges

An ideal machine learning I/O middleware should fulfill the following properties:

1. *Dataset Scalability*: The system should support datasets that are much larger than the aggregate storage/memory of the servers and should not limit the dataset size in any way.
2. *Node Scalability*: A solution should support systems with a few nodes but still scale well to many hundred nodes and take advantage of the additional storage/resources.
3. *Configuration Independence*: We do not want to place any assumptions on the storage configuration of the servers (e.g. if they include an SSD or not). The system should support a wide range of cluster environments and utilize their resources.
4. *Model Accuracy*: The data loading mechanism should not impact the model accuracy, for instance by changing the mini-batch sequences or using only a subset of files on some nodes. We generally want that the dataset is perfectly shuffled in each epoch, i.e. the probability to access a sample at any epoch offset should be $\frac{1}{F}$ (where F is the total number of samples).
5. *Ease of Use*: The solution should be easy to integrate into existing machine learning codebases and require only minimal changes by the user.

There are existing solutions, but they fail to meet all of these requirements at the same time. We briefly describe these solutions and their shortcomings. A more detailed description can be found in the referenced sources. The results of our analysis are summarized in Table 2.1.

- **tf.data with `shuffle`/`prefetch`:** In this approach, the tf.data API is used to build TensorFlow input pipelines. The `Dataset.shuffle()` transformation is used to randomly shuffle the input data and the

`Dataset.prefetch()` transformation creates a background thread and an internal buffer to prefetch elements according to the reference string.

As all nodes always fetch all samples from the PFS in this approach, it scales poorly when using many nodes for training. Furthermore, it only uses the RAM for prefetching and therefore does not make use of all the available resources depending on the environment. The user has to pass a buffer size to the `shuffle` call which controls the size of the buffer that is used for the sampling procedure. Therefore, the parameter choice can influence the training accuracy and perfect shuffling is only guaranteed if the parameter is chosen large enough.

- **Double Buffering:** As with tf.data, a buffer is filled according to the next accesses. The whole file list is shuffled every epoch and the prior knowledge (because the state of the PRNG is known) of the whole access string is used for prefetching. This approach is for instance used by PyTorch's `torch.utils.data.DataLoader` when `shuffle` is set to true or `RandomSampler` is passed (which is equivalent to setting `shuffle` to true), but the size of the prefetch window is only set to 2 times the number of loader worker processes[5]. As a proof of concept, we implemented a sampler for the Deep500 [5] benchmarking framework that also uses this strategy with a configurable prefetch window[6].

  This approach does not change the accuracy because the dataset is fully shuffled every epoch, but suffers from the same limitations as tf.data otherwise.

- **DeepIO:** This I/O framework uses RDMA-assisted in-situ shuffling, input pipelining, and entropy-aware opportunistic ordering to improve I/O performance when training deep neural networks. All of the data is exposed for RDMA transfer and the nodes use RDMA (remote direct memory access which allows direct memory access between different systems) to form the shuffled mini batches. In the entropy-aware opportunistic ordering mode, individual servers decide independently which elements constitute the next mini batch. Only the elements that are loaded into the in-memory storage buffers are considered when forming the batches. Cross-entropy is used to evaluate the impact of this mechanism on the randomization level of an input sequence.

  In this mode, model accuracy depends on the ratio of the shuffling memory size to the one of the whole dataset and perfect shuffling is only achieved when this ratio is 1. Furthermore, DeepIO only uses memory for prefetching and does not utilize other available resources.

[5] https://github.com/pytorch/pytorch/blob/a0dc36e501d336c92c10462057716f0355a0108c/torch/utils/data/dataloader.py#L745

[6] https://spclgitlab.ethz.ch/clairvoyant_prefetching/scripts/-/blob/master/MultiProcessSampler/multiprocesssampler.py

- **Parallel Data Staging:** To allow training on 3.5TB of data with 1,024 nodes, Kurth et al. [28] used an MPI-based staging script for distributing the files to the nodes before training. The dataset is divided into disjoint pieces that are accessed by one rank only. Each host uses multiple threads for reading in parallel and point-to-point MPI messages are used to distribute the files among the SSDs of the individual nodes.

  Because the whole dataset did not fit on the SSDs in their setup, each node only trained with a subset of the data which decreases model accuracy. Moreover, their approach requires each server to be equipped with node-local storage and therefore does not support a wide variety of configurations.

- **LBANN Distributed Data Store:** In LBANN's distributed data store, each process is assigned a subset of the data to store in memory. The locally-cached samples are distributed to the node that requires them at each training step with non-blocking MPI communication. Data samples are either read during the first epoch (dynamic approach) or fully loaded before training (preloading), in which case no further PFS accesses are required during training.

  As the whole dataset has to fit into the aggregated node memory, this approach does not support arbitrarily sized datasets. Furthermore, it only uses RAM to store the samples and not any other resources that may be available.

- **Locality-Aware Data Loading:** In this approach, the nodes form a distributed cache and can exchange samples among each other. An individual node forms local batches by going through the global mini-batch sequence and extracting the samples that are cached locally from it. A load balancing mechanism is used to ensure a balanced distribution. Samples that are not in the distributed cache are loaded from the parallel file system. Although this mechanism generally changes the access sequence, they show that it does not influence accuracy. But when applying batch-normalization on node-local batches, the results will differ.

  Only RAM is used for the distributed cache, which limits the resource utilization for some environments. The method also assumes that enough free memory (such that the aggregated memory can hold a significant portion of the dataset) is available at each node, otherwise the approach will load most of the samples from the PFS.

Chapter 3

System Design

3.1 Performance Model

In this section, we present the performance model that we will use for designing a new architecture (Section 3.2) and to compare existing solutions (Section 5.2). The most important definitions are summarized in Table 3.1.

Variable	Unit	Definition
N		Number of nodes
c	MB/s	Compute throughput
b_c	MB/s	Network bandwidth for inter-node communication
b_{fs}	MB/s	Network bandwidth for communication with the PFS
p_j		Number of threads for prefetching to storage class j
d_j	MB	Capacity of storage class j
D	MB	Total local storage of a node
$r_j(p)$	MB/s	Random aggregate read throughput of storage class j (with p reader threads)
$w_j(p)$	MB/s	Random aggregate write throughput of storage class j (with p writer threads)
$t(\gamma)$	MB/s	Random aggregate read throughput (with γ clients) of the PFS
β	MB/s	Preprocessing rate
F		Number of files
s_k	MB	Size of sample k
S	MB	Size of the whole dataset
B		Batch size
E		Number of epochs
T		Number of iterations
B^h		Subset of samples that are processed at iteration h
$B^{h,i}$		Subset of samples that are processed by node i at iteration h
R		Access pattern of a node

Table 3.1: Summary of Most Important Definitions.

3.1.1 Environment

We have N nodes with IDs $\{1, \dots, N\}$, where a node i is characterized by:

- c [MB/s]: Compute throughput of the node for the neural network we are interested in. This number depends heavily on the neural network architecture, the hardware of the node (CPU, GPU, FPGAs, or specialized hardware like Tensor Processing Units), the data sample size, and the used software.

We model the compute throughput as MB/s because it directly relates I/O parameters and the throughput. If it is only known in terms of samples per second, c can be approximated by multiplying this value by the average file size of the dataset that was used for the benchmark. If the samples are resized to a uniform size during preprocessing, the original data size should be used for this calculation.

- b_c [MB/s]: Network bandwidth for communication among nodes.
- b_{fs} [MB/s]: Network bandwidth for communication with the parallel file system.
- p_j: Number of threads that are used for prefetching data to storage class j. We assume that there always is at least one thread for prefetching to the staging buffer (see below for the definition of a storage class and the staging buffer), i.e. $p_0 \geq 1$.
- d_0 [MB]: The capacity of the staging buffer.
- d_j [MB]: The capacity of storage class j. The total local storage of a node is therefore given by

$$D = \sum_{j=1}^{J} d_j \, .$$

We further assume that there exists some shared parallel file system with a larger capacity than the dataset size and a random aggregate read throughput $t(\gamma)$ [MB/s] as a function of γ, the number of reading servers. Modeling the capacity as a function of γ is motivated by the observation that the bandwidth of parallel file systems can vary a lot with different numbers of clients [13], which we have also confirmed in our benchmarking (see Section 5.1.3). Moreover, we assume that there is no network contention, i.e. communication is always possible with bandwidth b_c or b_{fs}, depending on the endpoints.

To account for the storage diversity present in today's datacenters (as discussed in Section 2.2), we introduce J different storage classes (where J can be an arbitrary number depending on the environment) which group storage that behaves similarly concerning the metrics we are interested in. More precisely, a storage class j is characterized by:

- $r_j(p)$ [MB/s]: Random aggregate read throughput as a function of the number of reader threads p.
- $w_j(p)$ [MB/s]: Random aggregate write throughput as a function of the number of writer threads p.

The modeling of the throughput as a function of p is motivated by the observation that for many storage devices (especially fast ones, e.g. SSDs or

RAM), a single thread is not able to fully utilize the bandwidth of the device, which we have confirmed in an experiment that is described in Section 5.1.2.

Introducing storage classes allows the usage of our model for miscellaneous environments with different storage configurations. A storage class can for example represent RAM, SSDs, HDDs, shared global burst buffers, or emerging non-volatile memory technologies like magnetic random access memory (MRAM), spin-transfer torque random access memory (STT-RAM), or phase-change memory (PCM) [34]. Storage class 0 always represents the staging buffer which is a (usually small) in-memory buffer that is shared with the used machine learning framework.

3.1.2 Data

We have a total of F files that are used for training where sample k has size s_k. To account for the size variability that is present in many commonly used machine learning datasets (e.g. ImageNet [14], Youtube-8M [2], or UCF101 [40]), we allow different file sizes, i.e. we can have $s_k \neq s_h$ for some $k \neq h$. The size of the whole dataset (that is iterated over in one epoch) is given by

$$S = \sum_{k=1}^{F} s_k \,.$$

We can have $S > D$ (i.e. larger datasets than the total local storage of some or all nodes), the only restriction is that the whole dataset fits on the parallel file system. In the subsequent analysis, we assume that the dataset is initially stored on the PFS.

Depending on the training pipeline, various data preprocessing and transformation steps (e.g. decompressing, randomly resizing and cropping images, adding noise, etc.) are applied before injecting the data into the neural network. We assume that the data is preprocessed at a rate β [MB/s].

3.1.3 Training

We consider batches of size B. One epoch (where we denote the total number of epochs with E) therefore consists of $T = \lfloor \frac{F}{B} \rfloor$[1] iterations.

At iteration h (with $1 \leq h \leq E \cdot T$), we process the batch consisting of samples $B^h \subseteq \{1, \ldots, F\}$ and node i processes the node-local batch $B^{h,i} \subseteq B^h$, where we denote with $b_i = |B^{h,i}|$ the number of batch items that are processed by node i. As every data sample is read exactly once within a given epoch, B^k with $rT \leq k \leq (r+1)T$ (for some $r \in \mathbb{N}$) are disjoint sets, which implies that the sets $B^{k,i}$ are also disjoint for different values of k.

[1] $T = \lceil \frac{F}{B} \rceil$ if we do not drop the last iteration

When using data parallelism, $\{B^{h,1}, \ldots, B^{h,N}\}$ is a partition of B^h. In this case, we generally have $b_i = b_j$ for all[2] $1 \leq i < j \leq N$, although schemes with different sizes per node have been suggested [10]. When employing model parallelism or pipelining, we have $B^{h,i} = B^h$ for all $1 \leq i \leq N$ and for hybrid parallelism, various distribution schemes are used [6]. For our analysis, we assume data parallelism. However, the described solution also works for model/hybrid parallelism with different distribution schemes.

Given R, the access pattern of node i (where $B_l^{h,i}$ denotes the l-th sample in the node-local batch of iteration h):

$$R = (B_1^{1,i}, B_2^{1,i}, \ldots, B_{b_i}^{1,i}, B_1^{2,i}, \ldots) . \tag{3.1}$$

Let $t_{i,f}$ be the time when node i consumes R_f, the f-th entry of R.

$$t_{i,f} = \max\left(\text{avail}_i(f), t_{i,f-1} + \frac{s_{R_{f-1}}}{c}\right) , \tag{3.2}$$

where $\text{avail}_i(f)$ is defined as the elapsed time from the start of the training until R_f is available in the staging buffer. Assuming ideal load balance among the threads, we have

$$\text{avail}_i(f) = \frac{\sum_{k=1}^{f} \text{read}_i(R_k)}{p_0} . \tag{3.3}$$

We define $\text{read}_i(k)$ as the time required to read the k-th file into the staging buffer with:

$$\text{read}_i(k) = \text{fetch}_i(k) + \text{write}_i(k) . \tag{3.4}$$

The term $\text{fetch}_i(k)$ represents the duration for fetching the data into memory and $\text{write}_i(k)$ the time required to apply the transformations and store the data in the staging pool. $\text{write}_i(k)$ does not depend on the data source and according to the performance model (assuming applying transformations and writing the data to the staging buffer can be pipelined):

$$\text{write}_i(k) = \max\left(\frac{s_k}{\beta}, \frac{s_k}{w_0(p_0)/p_0}\right) .$$

For fetching the data, node i has up to three choices:

1. Reading from the parallel file system, while $\gamma - 1$ other nodes are reading from it as well:

$$\text{fetch}_{i,00}(k) = \frac{s_k}{\min(b_{fs}, t(\gamma)/\gamma)} . \tag{3.5}$$

[2]Assuming B is divisible by N, otherwise the size on the last node usually differs

2. Reading from another node that currently caches the file in storage class j and uses p_j threads for prefetching with:

$$\text{fetch}_{i,1j}(k) = \frac{s_k}{\min(b_c, r_j(p_j)/p_j)} \,. \tag{3.6}$$

3. Reading from its local storage class j (assuming the sample is stored in class j) with:

$$\text{fetch}_{i,2j}(k) = \frac{s_k}{r_j(p_j)/p_j} \,. \tag{3.7}$$

3.2 Design

3.2.1 Motivation

We motivate our design by the following two observations:

Observation 1: Requirements for Optimal Caching/Prefetching: Any optimal prefetching and caching strategy in a scenario with a single disk and processor has to follow these four rules [9]:

1. **Optimal Prefetching:** Every prefetch should bring into the cache the next block in the reference stream that is not in the cache.
2. **Optimal Replacement:** Every prefetch should discard the block whose next reference is furthest in the future.
3. **Do No Harm:** Never discard block A to prefetch block B when A will be referenced before B.
4. **First Opportunity:** Never perform a prefetch-and-replace operation when the same operations could have been performed previously.

Observation 2: Access Frequency Distribution: Although each sample is read exactly once within a given epoch, the number of times a node will read a given sample during training (i.e. the access frequency) will vary. If we consider a fixed node and sample and denote with X_e the probability that the node will access the sample in the e-th epoch, we have $X_e \sim \text{Bern}(\frac{1}{N})$. The access frequency X of this file is then given by

$$X = \sum_{e=1}^{E} X_e \,.$$

As the X_e are independent Bernoulli random variables with the same success probability (when the dataset is completely shuffled every epoch), it follows that X is binomially distributed with parameters E and $\frac{1}{N}$, i.e. $X \sim \text{B}(E, \frac{1}{N})$. The mean is $\mu = \mathbb{E}[X] = \frac{E}{N}$ and the probability that the access frequency is greater than it by a factor δ (which means that the file will be accessed more often by the node) is given by

$$\mathbb{P}[X > (1+\delta)\mu] = \sum_{k=\lceil(1+\delta)\mu\rceil}^{E} \binom{E}{k} (\frac{1}{N})^k (\frac{N-1}{N})^{E-k}.$$

We are not interested in the probability of a given file, but the number of files that will be accessed more often by a node. This is the sum over all samples of the indicator function $\mathbb{1}_{X>(1+\delta)\mu}$. Because the expectation is a linear operator and $\mathbb{E}[\mathbb{1}_{X>(1+\delta)\mu}] = \mathbb{P}[X > (1+\delta)\mu]$, the expected value is given by

$$F \cdot \mathbb{P}[X > (1+\delta)\mu]\,. \tag{3.8}$$

For example, plugging $N = 4$, $E = 1{,}000$, $F = 10{,}000$ and $\delta = 0.1$ into Equation 3.8 gives us an expected value of around 323. Therefore, although each sample will be read 250 times on average by each node, we expect that around 323 samples will be accessed more than 275 times.

We use a Monte Carlo simulation of the sampling procedure to validate these results. With the same parameters as given before, we get the empirical distribution of access frequencies depicted in Figure 3.1. The number of

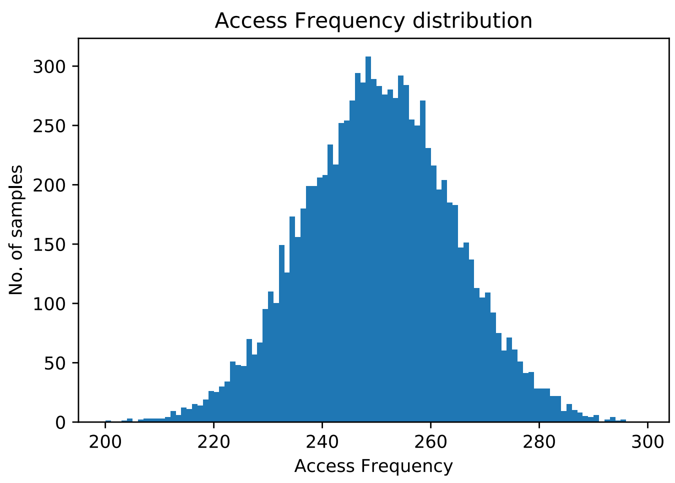

Figure 3.1: Monte Carlo Simulation Results.

samples that are accessed more than 275 times is 329, as expected by the calculations.

Every sample is accessed exactly E times globally. If some node accesses the sample more (or less) frequently, then some other node must access it less (or more) frequently. We prove the following two lemmas regarding the access frequency distribution that formalize this intuition:

Lemma 3.1 *If a node accesses a given sample $\lceil (1+\delta)\, \frac{E}{N} \rceil$ times, at least one other node will access the sample less than or equal to $\lceil (\frac{N-1-\delta}{N-1}) \frac{E}{N} \rceil$ times.*

Proof Assume for the sake of contradiction that every other node accesses the sample $\lceil (\frac{N-1-\delta}{N-1}) \frac{E}{N} \rceil + 1$ times for some $E, \delta \in [0, N-1]$ and $N > 1$. The total accesses to this file are then:

$$\begin{aligned} \lceil (1+\delta)\, \frac{E}{N} \rceil + (N-1) \left(\lceil (\frac{N-1-\delta}{N-1}) \frac{E}{N} \rceil + 1 \right) &\geq \\ (1+\delta) \frac{E}{N} + (N-1) \left((\frac{N-1-\delta}{N-1}) \frac{E}{N} + 1 \right) &= \\ (1+\delta) \frac{E}{N} + E - (1+\delta) \frac{E}{N} + N - 1 &= \\ E + N - 1 &> E\,. \end{aligned}$$

This is a contradiction to the fact described in Section 3.1.3 that every sample will be accessed exactly E times during training. □

Lemma 3.2 *If a node accesses a given sample $\lfloor (1-\delta)\, \frac{E}{N} \rfloor$ times, at least one other node will access the sample more than or equal to $\lfloor (\frac{N-1+\delta}{N-1}) \frac{E}{N} \rfloor$ times.*

Proof The proof is analogous to the proof of Lemma 3.1. We assume that every other node accesses the sample $\lfloor (\frac{N-1+\delta}{N-1}) \frac{E}{N} \rfloor - 1$ times. We then have for the total sample accesses:

$$\begin{aligned} \lfloor (1-\delta)\, \frac{E}{N} \rfloor + (N-1) \left(\lfloor (\frac{N-1+\delta}{N-1}) \frac{E}{N} \rfloor - 1 \right) &\leq \\ (1-\delta)\, \frac{E}{N} + (N-1) \left((\frac{N-1+\delta}{N-1}) \frac{E}{N} - 1 \right) &= \\ (1-\delta)\, \frac{E}{N} + E - (1-\delta)\, \frac{E}{N} - N + 1 &= \\ E - N + 1 &< E\,. \end{aligned}$$

Which is again a contradiction. □

3.2.2 Architecture

As we know the seed of the pseudorandom number generator, we fetch samples into the staging buffer according to the access string (i.e. R, as defined in Equation 3.1) of the node and therefore incorporate rule 1 into our design. After a sample of the staging buffer is read, the node will access it again at the earliest in the next epoch. Every sample that follows in the current epoch will be referenced for sure before the sample will be read again. To approximate the "Optimal Replacement", "Do No Harm" and "First Opportunity" rules, we therefore immediately drop samples from the staging buffer as soon as they are read by the machine learning framework, freeing up space for samples that will be referenced earlier with a high probability.

While the described strategy determines which samples to fetch into the staging buffer at which time, we still need to decide from where the samples should be read. For this, we use our analytical performance model that is described in Section 3.1. Among the possible options (where we assume for now that all nodes know which samples are cached by other nodes at any time, see Section 4.3.6 for details on how we ensure this in practice), a node chooses the option with the minimal fetch time, where the times are given by Equations 3.5 to 3.7.

We still need to define a strategy for the other storage classes. For this, we also need to decide on which samples to read in which order. If we assume the worst case in which a node always has to wait before consuming an entry of its access string R, we get for the total training time from Equations 3.2, 3.3 and 3.4:

$$t_{i,|R|} = \text{avail}_i(|R|) = \frac{\sum_{k=1}^{|R|} \text{read}_i(R_k)}{p_0} = \tag{3.9}$$

$$\frac{\sum_{k=1}^{|R|} \left(\text{fetch}_i(R_k) + \text{write}_i(R_k)\right)}{p_0} . \tag{3.10}$$

We want to fill the other storage classes in a way that minimizes this term. If we leave out everything that is fixed and does not depend on the strategy design, we get the following minimization problem:

$$\min \left(\sum_{k=1}^{|R|} \text{fetch}_i(R_k) \right) . \tag{3.11}$$

Plugging in the definition of fetch_i (given by Equations 3.5, 3.6 and 3.7), this becomes:

$$\sum_{k=1}^{|R|} \frac{s_{R_k}}{\max\left(\min(b_{fs}, t(\gamma)/\gamma), \min(b_c, r_{j_r}(p_{j_r})/p_{j_r}), r_{j_\ell}(p_{j_\ell})/p_{j_\ell}\right)} . \tag{3.12}$$

Where j_r and j_ℓ denote the fastest remote/local storage class of sample R_k respectively and $r_j(p_j)$ is the random aggregate read throughput of storage class j as defined in Section 3.1.1. When a file is not available at any remote node or locally, we define the respective throughput (i.e. $r_{j_r}(p_{j_r})$ or $r_{j_\ell}(p_{j_\ell})$) to be zero. If we denote with r_k the access frequency of the k-th file ($1 \leq k \leq F$), i.e. how often the index k appears in the string R and assume that we have a static system, i.e. the files are already in the storage classes in the beginning and no parameters change during execution, we can rewrite Equation 3.12 as a sum over all files:

$$\sum_{k=1}^{F} r_k s_k \frac{1}{\max\left(\min(b_{fs}, t(\gamma)/\gamma), \min(b_c, r_{j_r}(p_{j_r})/p_{j_r}), r_{j_\ell}(p_{j_\ell})/p_{j_\ell}\right)} . \quad (3.13)$$

Assuming $\forall j, k : s_k \approx s_j$, i.e. similar sized files, we can conclude:

1. Whenever r_k is large (in which case a node accesses a sample frequently), we generally want that $r_{j_\ell}(p_{j_\ell})$ is large, meaning the sample is cached in a fast local storage class.
2. Because $t(\gamma)/\gamma$ often stays constant or even decreases after a certain number of clients in practice, we generally want that γ, the number of concurrent PFS clients, is small, ensuring that PFS reads (if they are inevitable) are fast. Furthermore, $r_{j_r}(p_{j_r})$ should be large for samples where $r_{j_\ell}(p_{j_\ell})$ is small or zero, i.e. samples that are not stored locally at all or only in a slow storage class. Both requirements imply that the samples should be well distributed among nodes.

Because of Observation 2 of Section 3.2.1, we expect that r_k varies for different values of k. Furthermore, because of Lemmas 3.1 and 3.2 we expect that if r_k has a small value for a given node, there will be at least one other node where the value is large and vice versa.

Therefore, we use the access frequency r_k as the criterion for the fetching decision. A node will fetch the samples with the highest access frequency into its fastest storage class. Among the remaining files, the ones with the highest access frequency are chosen for the second-fastest storage class. This process continues until either no more samples are left, i.e. the whole dataset is cached or no more space is left, i.e. the dataset is larger than the aggregated node storage. Using the access frequency as the criterion ensures that both desired properties are fulfilled.

After we have determined which samples to read for a storage class, we still need to define the reading order. In our analysis so far we assumed a static system where all the samples are already fetched in the beginning. This is obviously a simplifying assumption as the samples become available during execution. For the order, we use again rule 1 of the requirements for optimal caching and prefetching. This means that for a given storage class, the fetching order is determined by the first access of the sample. The design of HDMLP, the **H**ierarchical **D**istributed **M**achine **L**earning **P**refetcher, is illustrated in Figure 3.2.

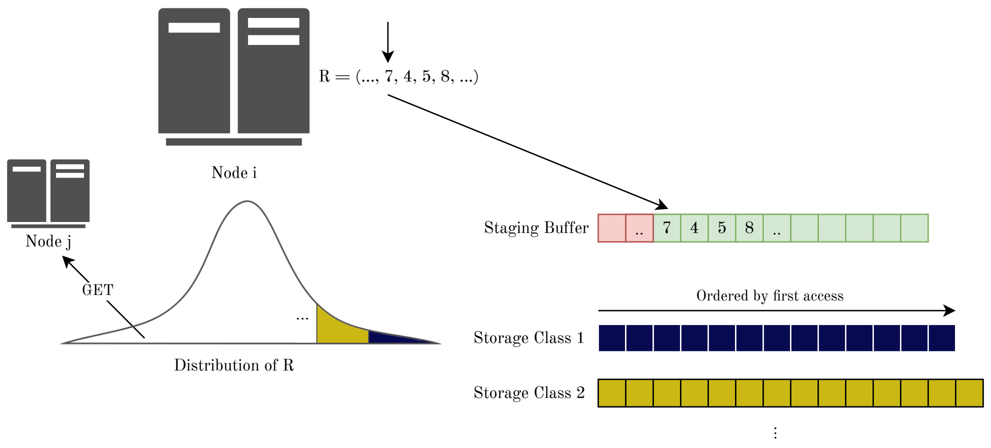

Figure 3.2: Visualization of the HDMLP Design.

Chapter 4

System Implementation

4.1 Performance Simulation

Based on our analytical performance model described in Section 3.1, we implement a performance simulation framework in Python to test and evaluate different data loading strategies. The framework allows to implement different policies and compare them against each other in various scenarios. The user can configure arbitrary scenarios, but we provide four pre-defined ones that cover a wide range of use cases:

1. $S < d_1$: In this scenario, the data fits into the first storage class (typically RAM) of each node. The scenario should not be particularly challenging for any strategy but is important nevertheless (as it occurs with small datasets or nodes that are equipped with a lot of memory).
2. $d_1 < S < D$: Here, the dataset is larger than the capacity of the first storage class but smaller than the aggregated storage of an individual node. This is an interesting scenario because the whole dataset can still be stored completely on an individual node but a strategy needs to utilize multiple storage classes to do so.
3. $D < S < N \cdot D$: With this setup, the data no longer fits into the aggregated storage of an individual node but is still smaller than the aggregated storage of all nodes. The setup is particularly challenging as collaboration between the individual nodes is required to minimize the number of PFS accesses.
4. $N \cdot D < S$: This scenario simulates a dataset that does not even fit into the aggregated storage of all nodes. While this should not happen too often in practice at the moment when using a large number of nodes for training, it is still interesting to see how individual strategies handle it. Furthermore, with the fast-growing size of machine learning datasets, it can become more common in the future and is already a

common scenario when working with moderately large datasets on a small cluster.

Moreover, we allow the testing and comparison of the strong scaling behavior of different policies. To do this, we run the same evaluation multiple times, changing the number of simulated nodes every time. We then report the execution times for the different number of nodes.

Besides enabling us to compare different policies against each other, the performance simulation also allows quantifying the impact of changes to the environment on the runtime of neural network training. For instance, it could be used when designing new cluster environments to analyze whether an investment into additional storage provides enough time savings for the intended workload.

4.1.1 Environment

The user can provide all the environment parameters that are described in Section 3.1.1. A sample configuration can be found in Listing 4.1.

```
N = 4 # Number of nodes
c = 100 # Compute throughput [MB/s]
beta = 200 # Preprocessing rate [MB/s]
b_c = 10000 # Network bandwidth for node communication [MB/s]
b_fs = 2000 # Network bandwidth for FS communication [MB/s]
p = {0: 2, 1: 2, 2: 2} # Number of prefetcher threads per storage level
d = {0: 1024, 1: 50*1024, 2: 100*1024} # Capacity per storage level [MB]
t = {1: 66, 2: 86, 4: 146, 8: 326, 16: 324, 32: 324, 64: 324, 128: 324}
    # PFS random read bandwidth [MB/s]
# Storage classes and their read throughput per number of threads [MB/s]
storage_classes = {
    0: { # RAM (staging pool)
        1: 16550,
        2: 21164,
        3: 21186,
        4: 21415
    },
    1: { # RAM (prefetch buffer)
        1: 16550,
        2: 21164,
        3: 21186,
        4: 21415
    },
    2: { # SSD
        1: 66,
        2: 86,
        3: 129,
        4: 146
    }
}
```

Listing 4.1: Environment Configuration Example.

If no random read bandwidth is specified for a number of reading servers γ (that is used during simulation), the value is linearly interpolated, i.e.:

$$t(\gamma) = \begin{cases} t(\gamma_0), & \text{if } \gamma_0 > \gamma \\ t(\gamma_1), & \text{if } \gamma_1 < \gamma \\ t(\gamma_0) + (\gamma - \gamma_0)\frac{t(\gamma_1) - t(\gamma_0)}{\gamma_1 - \gamma_0}, & \text{otherwise,} \end{cases} \tag{4.1}$$

where γ_0 is the largest key that is smaller than γ (if it exists, the minimum key otherwise) and γ_1 the smallest key that is larger than γ (if it exists, the maximum key otherwise).

4.1.2 Data

There are two options for the simulation of the sample size distribution. The sizes can be generated from an arbitrary distribution or the path to an existing dataset can be provided, in which case the file sizes and the number of samples from this dataset are used. Listing 4.2 shows an example where 10,000 samples are generated from a normal distribution with $\mu = 0.027$ (MB) and $\sigma = 0.01$.

```
dataset = {
    'synthetic': True,
    'synthetic_params': {
        'distribution': scipy.stats.norm,
        'loc': 0.027,
        'scale': 0.01,
        'samples': 10000,
        'min': 0,
        'max': None
    },
    'dataset_params': {
        'path': None
    }
}
```

Listing 4.2: Data Configuration Example.

An arbitrary object that provides a `rvs(loc, scale, size)` function and returns a list or NumPy array of `size` samples can be passed as distribution.

4.1.3 Training

Besides the number of epochs and batch size, the user can also specify if the last iteration should be dropped (when the number of samples is not divisible by the batch size). Furthermore, it can be specified (`node_distr_scheme`) how the node-local batches $B^{h,i}$ are formed from the batch B^h. Currently, the following schemes are supported:

- `uniform`: A batch B^h is partitioned into disjoint sets that are distributed among the nodes. The sizes of the node-local batches are identical except for the last node which gets the remaining samples if $|B^h|$ is not divisible by N.

- `whole`: Every node gets the whole batch B^h, i.e. $\forall i : B^{h,i} = B^h$.
- `imbalanced`: Half of the nodes get node-local batches of size $2\lceil\frac{|B^h|}{N}\rceil$, whereas the size is $\frac{1}{2}\lceil\frac{|B^h|}{N}\rceil$ for the other half.

```
E = 10 # Number of epochs
B = 100 # Batch size
drop_last_iter = True
node_distr_scheme = 'uniform'
```

Listing 4.3: Training Configuration Example.

4.1.4 Simulation

As a last step, the user has to specify if the generated plots should be aggregated per batch/iteration (instead of reporting values per sample) and he has to provide a dictionary of policies that should be used for the simulation. Policies can have additional parameters in their constructor, which are specified via `opts`.

```
aggregated_stats = True
seed = None
policies = {
    "Lower Bound": {'class': PerfectPrefetcher, 'opts': {}},
    "Naive": {'class': BasePolicy, 'opts': {}},
    "HDMLP": {'class': FrequencyPrefetcher, 'opts': {'in_order': True}},
    "StagingPoolPrefetcher": {'class': StagingPoolPrefetcher, 'opts':
        {}},
    "DeepIO (Ordered)": {'class': DeepIOPolicy, 'opts': {'
        opportunistic_reordering': False}},
    "DeepIO (Opportunistic)": {'class': DeepIOPolicy, 'opts': {'
        opportunistic_reordering': True}},
    "Parallel Data Staging": {'class': ParallelStagingPolicy, 'opts':
        {}},
    "LBANN In Memory Store (Dynamic)": {'class':
        LBANNInMemoryStorePolicy, 'opts': {'preloading': False}},
    "LBANN In Memory Store (Preloading)": {'class':
        LBANNInMemoryStorePolicy, 'opts': {'preloading': True}},
    "Locality-Aware Data Loading": {'class':
        LocalityAwareDataLoadingPolicy, 'opts': {}}
}
```

Listing 4.4: Simulation Parameters Example.

After all parameters are specified, a `Dataset` object is created and is passed with all the other parameters to a `PolicyExecutor` object, which executes the simulation and returns a `Statistics` object, providing different information and plots about the simulated execution. We will now describe these different parts in more detail:

Dataset

This class is responsible for generating the file sizes that will be used in the simulation (based on the configuration) and generating the node-local batches, i.e. which samples are accessed in which order by the different nodes. If the user configured a synthetic file size distribution, the class samples from this distribution. Otherwise, the file sizes from the specified path are read and used.

PolicyExecutor

The `PolicyExecutor` class takes policies (see below) and the other parameters (described above) as input, runs the simulation with the provided parameters and returns the statistics about the execution. We simulate the bulk-synchronous nature of consistent model methods [6], meaning we take the maximum run time across all nodes per batch and set the start time for the next batch to this value for all nodes (even if they finished the previous batch earlier).

Policies

We provide the `BasePolicy` class as a base class for policies. The class is intended to be extended by the specific policies and provides various helper utilities for creating own policies. If the class is instantiated directly, it simulates loading the data from the PFS without prefetching or caching and can therefore be used as a naive baseline.

The `request` method takes a batch number, batch offset, node ID, and request time as input and returns the time when the request is fulfilled. To do that, the method checks whether the sample is available locally or on another node and chooses the available option with the minimum loading time among the available options. The information of whether a file is available at a given time comes from the specific policies that extend the class. This is accomplished by the two methods `fill_buffers` and `cleanup_buffers` that should be overwritten by user-implemented policies. In these methods, the prefetching/caching logic of a policy can be implemented, i.e. what samples in which buffers are fetched/dropped at a given time. For the implementation of this logic, the policies can access the complete information about the system (for instance what files are currently available at which node) at any time. While this is not always realistic in practice (as some of the global information is not available or only with additional communication overhead), it allows modeling and comparison of arbitrary policies. If a user wants to only implement policies that can be directly translated to real algorithms, the information that is not available in practice can simply be ignored. The reference string is provided to the policies by the framework.

However, some policies change it to implement different types of optimizations. These changes are fully supported by the framework.

We provide an implementation for the following policies:

- `PerfectPrefetcher`: A simple policy that simulates a scenario where no stalls occur, i.e. all samples are available immediately. While this is not viable for many configurations, it provides a lower bound (with regards to the data loading mechanism) for the execution time.
- `StagingPoolPrefetcher`: Simulates a prefetching scenario with a staging pool that is filled according to the reference string. The data is fetched from the fastest available location and samples are dropped immediately after they are consumed. If the policy is instantiated directly, this therefore corresponds to prefetching from the parallel file system and is a performance simulation of the two approaches tf.data with `shuffle/prefetch` and double buffering, as described in Section 2.3. Besides direct instantiation, the policy can also be extended by other policies, allowing the logic of prefetching according to the reference string and choosing among the best options (which is a shared theme among many policies) to be reused.
- `DeepIOPolicy`: This policy simulates the performance of DeepIO (see Section 2.3 for details). It extends the `StagingPoolPrefetcher` and can simulate DeepIO in ordered or opportunistic mode.
- `ParallelStagingPolicy`: The class simulates the behavior of parallel data staging, as described in Section 2.3. Like the `DeepIOPolicy` in opportunistic mode, the policy changes the provided reference string, because only samples that are available locally are considered for training.
- `LBANNInMemoryStorePolicy`: With this policy, we simulate the LBANN distributed data store that was described in Section 2.3. Because the store only supports datasets that are smaller than the aggregated node memory, the policy throws an exception if the dataset size exceeds this value in a configuration. The exception is caught by the `PolicyExecutor` and no statistics for the run are produced in this case. We implemented both the dynamic approach and the setup where the dataset is completely loaded before training, which is controlled by the parameter `preloading`.
- `LocalityAwareDataLoadingPolicy`: With this policy, we simulate locality-aware data loading (see Section 2.3 for details). As the mechanism also changes the reference string, we reorder the batches in the beginning in correspondence with the logic that is described in the paper.

- `FrequencyPrefetcher`: The policy, instantiated with `in_order` set to true (which controls whether the order per storage class is according to the first reference or according to the access frequency) corresponds to the prefetching and caching scheme of HDMLP as described in Section 3.2.

4.2 HDMLP - Python Frontend

We implemented the design that is described in Section 3.2 as a flexible and easy-to-use I/O framework. The implementation consists of around 2,000 lines of C++ code and 500 lines of Python code. We use the `ctypes`[1] library to interact with our framework from Python. The overall system architecture is depicted in Figure 4.1, we describe the different parts in Section 4.2 (Python Frontend), 4.3 (C++ Backend), and 4.4 (Data Augmentation) in more detail. The corresponding source code is available on GitHub[2].

4.2.1 Python HDMLP Library

The Python HDMLP library provides a Python interface to the HDMLP C++ library and allows easy integration of the library into existing Python projects, for instance as a data loader/dataset (as described in the next section). It provides the `Job` class, which is meant to represent the execution of a machine learning job (e.g., training or validation on a specific dataset). Note that there is not necessarily a one-to-one correspondence between a job and a Python process. A machine learning framework can submit multiple jobs at the same time (e.g., one job for the training dataset and one job for the validation dataset), which is supported by the framework.

When creating a new HDMLP job, the user has to specify:

- `dataset_path`: Path to the dataset.
- `batch_size`: The global batch size that will be used for this job.
- `epochs`: The number of epochs.
- `distr_scheme`: The distribution scheme that is used to distribute a batch among nodes. At the moment, we only support the `uniform` distribution scheme (as described in Section 4.1.3), but other schemes could be easily implemented if needed. The required modifications for new schemes are explained in the sampler description.
- `drop_last_batch`: Whether the last batch should be dropped or not if the dataset size is not evenly divisible by the global batch size.

[1] https://docs.python.org/3/library/ctypes.html
[2] https://github.com/spcl/hdmlp

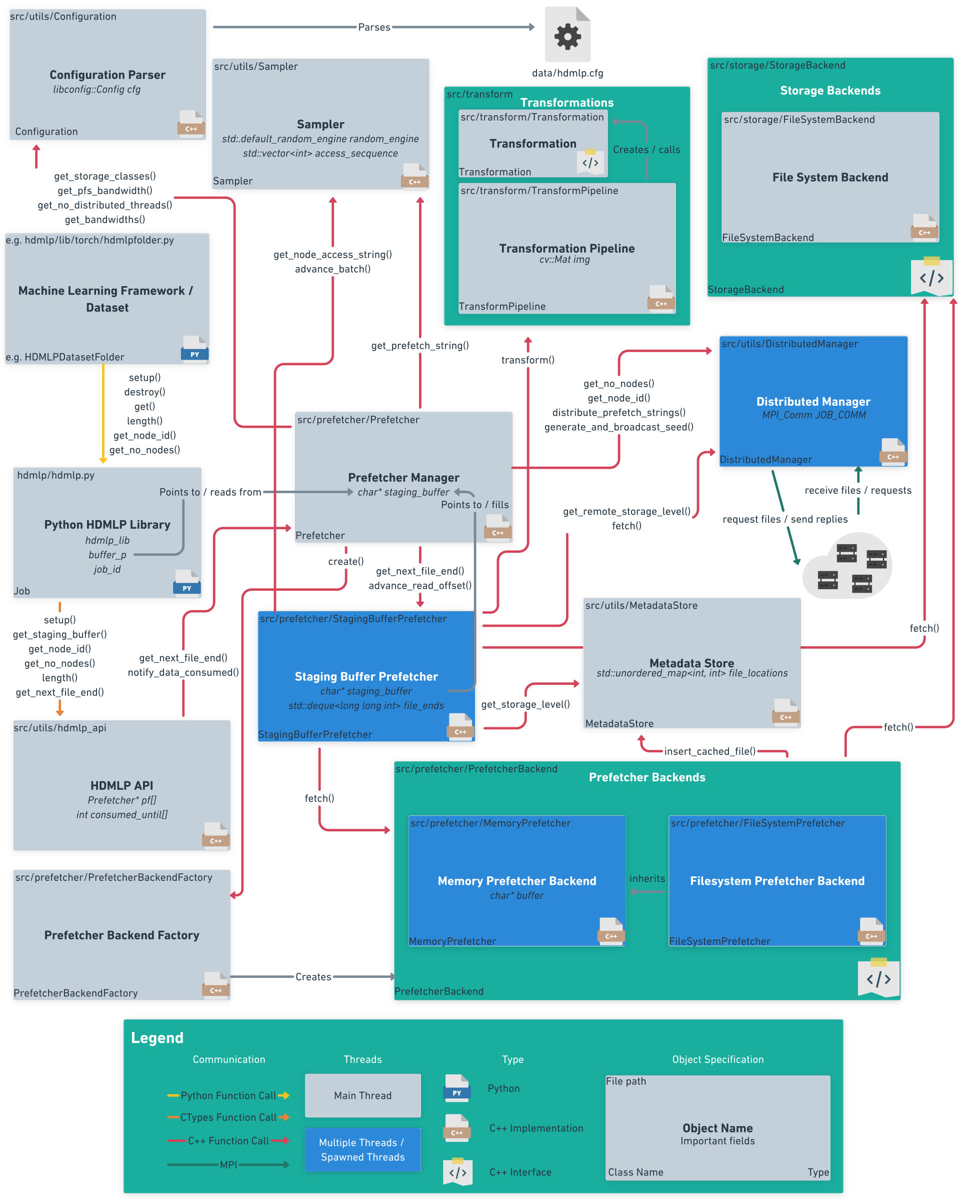

Figure 4.1: HDMLP System Architecture.

- `transforms`: An optional list of HDMLP's integrated data transformations (see Section 4.4 for more details) to apply. When no transformations are specified, the samples are returned unmodified.
- `seed`: An optional seed value for the initialization of the pseudo random number generator that is used for data shuffling. If no seed is provided, the framework will generate and broadcast one on its own.
- `config_path`: The user can overwrite the default `libhdmlp` (HDMLP's C++ backend) configuration path (`data/hdmlp.cfg`) if needed.
- `lib_path`: If the `libhdmlp` dynamic library is installed in a non-default location, the path can be specified with this parameter. When the provided setup script is used for the installation, the parameter is not needed as the framework will find the library automatically.

After the creation of a job, the `setup` method has to be called for the start of the prefetching procedure. This method initializes the C++ library and sets the `buffer_p` attribute of the Python library. The attribute contains a pointer to the staging buffer which is circularly filled by `libhdmlp`, allowing zero-copy access to the prefetched samples.

When a job is finished, `destroy` should be called which cleans up the internal state of the C++ library and deallocates data.

The most important method is `get`, which returns one sample at a time (when no HDMLP transformations are applied) or a batch of samples including the labels. To do that, it gets the end of the sample/batch by a call into the C++ library (and knows the beginning because data is laid out consecutively in the buffer and the previous file end is stored internally). `libhdmlp` always stores the labels in front of the actual data and labels are separated from the data by zero bytes. This allows decoding of the labels without storing any additional length information.

The Python frontend also provides some convenience methods like `length`, `get_node_id`, or `get_no_nodes` but is kept rather small otherwise.

4.2.2 Machine Learning Framework Integration

For the integration of the framework with PyTorch, we provide three dataset implementations that closely follow the `DatasetFolder`, `ImageFolder`, and `ImageNet` datasets from Torchvision[3] and a data loader implementation. `HDMLPDataLoader` is a simple data loader that takes a dataset as input and provides an iterator that returns batches as a tensor. If the framework is used without its internal data transformations (and therefore returns one sample at a time on which data transformations may be applied by the used dataset

[3] https://pytorch.org/docs/stable/torchvision/datasets.html

implementation), the batches are formed by the data loader itself. Otherwise, the data loader simply returns the batches that were already formed by HDMLP. We decided to form the batches in the framework (instead of returning individual samples) when using the integrated transformations because our experiments showed that the creation of the batches (for which every individual sample has to be copied) in the data loader can become a bottleneck and increase stall times for large batch sizes. With our implementation, samples are never copied in memory after HDMLP places them into the staging buffer.

Using these reference implementations, we show that HDMLP can be easily integrated into existing machine learning frameworks. Our data loader implementation is quite short with 60 lines of code and our datasets required only a few changes of the existing Torchvision datasets. The implementations allow us to use HDMLP for existing PyTorch code with minimal changes. For instance, if some existing distributed PyTorch code initializes the dataset/data loader like this:

```
dataset = ImageFolder(data_dir, data_transforms)
dsampler = DistributedSampler(dataset, num_replicas=n, rank=node_id)
dataloader = DataLoader(dataset, batch_size, sampler=dsampler)
```

Listing 4.5: Example PyTorch Dataset/Data Loader Initialization.

This becomes (when continuing to use Torchvision's transformations, which is also supported by our implementation):

```
job = Job(data_dir, batch_size, num_epochs, 'uniform', drop_last_batch)
dataset = HDMLPImageFolder(data_dir, job, data_transforms)
dataloader = HDMLPDataLoader(dataset)
```

Listing 4.6: Example HDMLP PyTorch Dataset/Data Loader Initialization.

No other changes are required.

4.3 HDMLP - C++ Backend

The backend of HDMLP is written in C++ and built as a shared library called `libhdmlp`. The different parts of the library are described in the following sections.

4.3.1 HDMLP API

This is the interface that is used by the Python process to interact with the C++ library. We declare the functions of the API with `extern "C"` such that the compiler does not add extra mangling information to the symbols and we can call the functions using `ctypes`.

The interface keeps track of all `Prefetcher` objects (where each object represents a job) and is responsible for the allocation/deallocation of them (when the Python library instructs it to do so with a `setup/destroy` call). Besides that, the interface mainly provides certain required functions of the `Prefetcher` class to the Python library.

4.3.2 Prefetcher Manager

The `Prefetcher` class represents a job with all of its internal state, allocated data, and objects. It is responsible for the allocation/deallocation of all other objects and provides all the functions that are needed by the Python library.

It first gets the parsed config and initializes the storage backend, metadata store, distributed manager, and optionally the transformation pipeline. If the user did not provide a seed, the distributed manager is then used to generate and broadcast one, therefore ensuring that the shuffling of the file IDs will result in the same sequence on all nodes. Then, a sampler object is instantiated with the seed and the prefetch string (i.e. which samples should be fetched into which storage class in which order) is generated for the node. The prefetch strings are distributed to the other nodes by the distributed manager, which allows the nodes to infer when a sample will be available at a remote node. Note that instead of distributing the prefetch strings, we could also compute them locally for all nodes in the beginning as all the required information (the seed and the node ID of the other nodes) is available. However, we decided to distribute them such that the computation is done only once for every node. Finally, all the prefetcher backend objects are allocated and the threads for prefetching and distributed caching are created. We will now explain these steps and the involved classes in more detail.

4.3.3 Configuration Parser

We use the `libconfig`[4] library to process the configuration file.

```
b_c = 10000
b_fs = 2000
distributed_threads = 2
checkpoint = true
checkpoint_path = "/tmp"
storage_classes: ( {
        capacity: 1024 # [MB]
        threads: 2
        bandwidth: (
            { threads = 1; bw = 16550; },
            { threads = 2; bw = 21164; },
            { threads = 3; bw = 21186; },
            { threads = 4; bw = 21415; }
        );
```

[4]http://hyperrealm.github.io/libconfig/

```
},
{
        capacity: 2048 # [MB]
        backend: "memory"
        threads: 2
        bandwidth: (
            { threads = 1; bw = 16550; },
            { threads = 2; bw = 21164; },
            { threads = 3; bw = 21186; },
            { threads = 4; bw = 21415; }
        );
},
{
        capacity: 10240 # [MB]
        backend: "filesystem"
        backend_options: {
            path = "/tmp/hdmlp";
        }
        threads: 2
        bandwidth: (
            { threads = 1; bw = 192; },
            { threads = 2; bw = 399; },
            { threads = 3; bw = 599; },
            { threads = 4; bw = 637; }
        );
});
pfs_bandwidth: (
    { processes: 1; bw: 66; },
    { processes: 2; bw: 86; },
    { processes: 4; bw: 146; },
    { processes: 8; bw: 326; },
    { processes: 16; bw: 324; },
    { processes: 32; bw: 324; },
    { processes: 64; bw: 324; },
    { processes: 128; bw: 324; }
);
```

Listing 4.7: Example HDMLP Configuration File.

An example configuration file is depicted in Listing 4.7. The supported parameters are:

- `b_c`: Corresponds to b_c from the performance model, i.e. the network bandwidth for communication among nodes.
- `b_fs`: Corresponds to b_{fs} from the performance model, i.e. the network bandwidth for communication with the PFS.
- `distributed_threads`: The number of threads that should be launched for serving files to other nodes, see "Distributed Manager" for details.
- `checkpoint`: For large datasets, the initial parsing of the file names, labels, and file sizes can take some time as it involves many metadata operations (i.e. `readdir()` and `stat()`). We therefore provide the option to store this information after the first parsing which speeds up initialization of further runs on the same dataset.

- `checkpoint_path`: If `checkpoint` is set to `true` and this option is not specified, the metadata information is stored in the root folder of the dataset. Because datasets are located on read-only locations in some setups (or no new files should be created in the root folder for other reasons), we provide the option to override this path.
- `storage_classes`: A list of all the configured storage classes. The first item in the list always represents the staging buffer. For the other elements of the list, we assume that the user provides them according to his usage preferences, meaning a storage class that is earlier in the list should be used more often. This generally means that they are ordered according to the read bandwidth. If a user has other usage preferences for some reason (e.g. if one class is backed by a shared medium and he wants to avoid intensive usage), he is free to change the order. The configuration of one storage class consists of:
  - `capacity`: How much space of the storage class should be used (specified in MB). This parameter allows setups where we only use a subset of the available storage.
  - `backend`: Which prefetcher backend (see below for more details) to use (except for the staging buffer, as the backend is fixed). At the moment, we support `memory` for RAM and `filesystem` for POSIX filesystem access, however other backends can easily be integrated into our solution.
  - `backend_options`: We support configurable prefetcher backends with additional parameters that change their behavior. Currently, the only used parameter is `path` (which controls the path that will be used to store temporary files) for the `filesystem` backend.
  - `threads`: The number of threads to use for prefetching to this storage class, i.e. p_j from the performance model.
  - `bandwidth`: The configuration of bandwidth per number of reading threads, i.e. $r_j(p)$ from the performance model. If no value for the configured value of p is provided, it will be linearly interpolated.
- `pfs_bandwidth`: The random aggregate read throughput per number of reading servers, i.e. $t(\gamma)$ from the performance model. If the value for a specific number of nodes is not specified, it will be linearly interpolated according to the formula in Equation 4.1.

4.3.4 Storage Backends

A storage backend implements the operations that are used to access the source files. For greater flexibility, we provide the generic `StorageBackend` interface with the functions `get_label(file_id)`, `get_length()`, `get_file_size(file_id)` and `fetch(file_id, *dst)` that any concrete implementation needs to implement. At the moment, we provide the `FileSystemBackend` implementation which supports datasets that are accessed via the file system and can therefore be used for many different kinds of storage devices/systems. However, other storage backends that for instance directly fetch HDF resources or access data in an object storage over specialized APIs could easily be implemented.

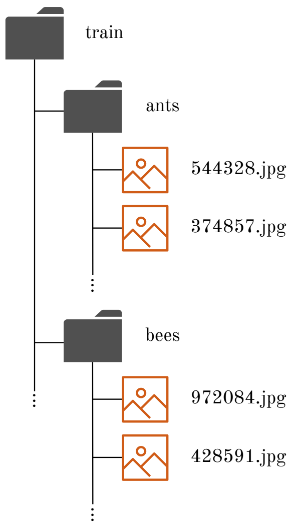

Figure 4.2: Dataset Folder Structure for the `FileSystemBackend` Class.

Our `FileSystemBackend` implementation assumes that there exists a subfolder for every label, containing all the samples with this label, as depicted in Figure 4.2. Note that the name of the subfolder does not necessarily need to correspond to the name of the label, as transformations to the name can still be applied by the corresponding dataset class. This is illustrated in our `HDMLPImageNet` dataset, where `libhdmlp` returns the label id. The data loader parses an additional metadata file to allow translation between these IDs and the class names if needed.

As we are referring to files by their IDs (where the maximal ID is given by the number of samples - 1, i.e. `get_length() - 1`) in all the other parts of the library, every storage backend implementation needs to ensure that the same ID maps to the same file on every node (otherwise, wrong files would be returned when requesting a file by its ID from another node). In our `FileSystemBackend` implementation, we sort the files by their label and file name (as `readdir` does not guarantee any order for the files[5]) before assigning the IDs to ensure this property holds.

[5] http://man7.org/linux/man-pages/man3/readdir.3.html

4.3.5 Metadata Store

The metadata store is responsible for keeping a catalog of locally cached files. Furthermore, it can return for a given local and remote storage level the order of the three options (local storage level, remote storage level, and PFS), where the options are sorted according to the read bandwidth which is given by Equations 3.5 to 3.7 of the performance model. Note that we always assume the worst case for the number of PFS clients, i.e. $\gamma = n$. While this can be overly pessimistic in certain scenarios, it will never result in more reads to the shared PFS than a hypothetical implementation that is always aware of the exact number of PFS clients.

To keep track of the locally cached files, a pointer to the metadata store is passed to the different prefetcher backends, which need to call `insert_cached_file(storage_level, file_id)` after they fetched a file.

4.3.6 Distributed Manager

We use the `DistributedManager` class to manage the MPI connections with the other hosts and for all tasks involving multiple hosts. To support the simultaneous usage of HDMLP with other libraries that use MPI internally (e.g., Horovod), we first check if MPI is already initialized and only initialize it if the check fails. Furthermore, we create a private MPI communicator for the library. Because multiple threads call MPI functions in our implementation, we also check if the implementation supports the thread support level `MPI_THREAD_MULTIPLE` and throw an error if it does not.

The most important functions of the class are:

- `generate_and_broadcast_seed()`: If the user did not provide a seed, node 0 generates one and broadcasts it to all nodes, therefore ensuring that the same global access sequence is generated on every node.
- `distribute_prefetch_strings(local_prefetch_string, storage_class_ends, num_storage_classes)`: As described previously, the prefetch strings (i.e. which samples are fetched in which order) are distributed in the beginning such that every node knows when a file will be available at another node. As the sizes of the prefetch strings can vary per node, we first use `Allreduce` to get the maximum string length among the nodes. Every node then allocates a buffer that can hold the longest string, the number of elements per storage class and the number of storage classes that are used on the node (as this number can be different in some edge cases where the whole dataset fits into the aggregated storage of a single node, but because we are dealing with variable sized files and the placement order differs among hosts, some nodes use one storage class more than others). This information

is put into the buffer by every node and we use `Allgather` such that the information is communicated with every rank.

We then proceed to parse the received information (i.e. reconstruct what storage class will hold which samples) and every node stores for every sample in which remote storage class it is available, on which rank, and the file's offset (i.e. how many files in the same storage class will be fetched before the sample). This information will be used in `get_remote_storage_class` to decide whether a file is available at a remote node at different time points. If there are multiple nodes that cache a sample, we choose the one where it is stored in the storage class with the lowest index and offset to achieve a good distribution of the requests and therefore avoid contention.

- `get_remote_storage_class(file_id)`: The staging buffer prefetcher calls this function when deciding on the source of a prefetch operation. When no other node caches a file, we can tell for sure that it is not available at any remote node. Otherwise, the availability depends on the prefetch progress of the node that we are considering, which generally is not known. To avoid additional metadata or progress messages, we assume that the average file size of the prefetched files is similar among nodes and the progress of different nodes does not differ too much. Based on this assumption, we decide that a file with a given offset (that we know from the distribution step) is available in storage class j on a remote node if one of the following properties holds:

  1. The storage class j does not exist locally. This corresponds to the edge case that we described earlier. As the storage class j on the remote node only contains very few entries in this case, we can assume that the file is available.
  2. The current prefetch progress for storage class j on the local node is sufficiently larger (where the required difference is controlled by `REMOTE_PREFETCH_OFFSET_DIFF`) than the offset of the file.
  3. Prefetching for storage class j on the local node has finished.

  Note that there will not be an error if this heuristic fails and indicates that a file is available at another node when it is not as these cases are detected and handled by the framework. However, we want to minimize such false positives as they hurt performance. We confirmed in different experiments that the heuristic performs well in practice and has very few false positives.

- `serve()`: This function is used for serving samples to other nodes and we provide the configuration parameter `distributed_threads` to control the number of threads that execute it. These threads listen for messages with the tag REQUEST_TAG which contain the requested file ID and the answer tag (see the description of fetch below for details). If a file is not available locally but we still get a request for it (which should occur infrequently as we are only sending requests when the file is available at a remote node with high probability, see get_remote_storage_class for details), we send back an empty MPI message to indicate the issue. Otherwise, we send back the requested file. Note that although we are dealing with variable sized files, we do not need to do any probing as every node knows all the file sizes and can therefore allocate a buffer with the correct size without additional overhead.

- fetch(file_id, *dst, thread_id): Requests a file from a remote node and stores it at dst. The function first checks at which node the sample is stored and then sends a request, consisting of the file ID and thread_id + 1 to this node. thread_id + 1 will be used as the message tag for the reply, the nodes therefore listen for messages with this tag after sending the request. This allows multiple threads to send requests to the same node in parallel without mixing up the responses. When we get back an empty response (indicating that the file is not available yet), we return false to the caller to ensure that it is aware of the failure. The caller (staging buffer prefetcher) will then request the sample from a different source.

4.3.7 Sampler

In our implementation, the Sampler class is responsible for shuffling the dataset and generating the prefetch strings for the individual storage classes. To do that, it first needs to calculate the access frequency for all the samples without advancing the state of the pseudo random number generator (as the actual training would use a different access frequency distribution otherwise). This is implemented in get_access_frequency by saving and restoring the state of the random engine.

As described earlier, the framework has built-in support for using different schemes when distributing batches among nodes. At the moment, we provide an implementation for the uniform scheme but arbitrary schemes can be implemented in the function get_node_access_string_for_seq (the mapping of distribution scheme strings to IDs is stored in the DISTR_SCHEMES map of the Python library) by simply returning the access string for a given node ID according to the desired scheme.

In get_prefetch_string, we use the access frequency information to generate the prefetch strings for the individual storage classes. First, we order the samples by the access frequency. Then, we analyze how many samples fit into the individual classes and finally, we order the individual classes by the first access.

The sampler object is also passed to the staging buffer prefetcher, which uses it to prefetch according to the reference string.

4.3.8 Staging Buffer Prefetcher

The `StagingBufferPrefetcher` class fills the staging buffer in a circular manner, meaning we wrap around as soon as the end is reached. There are two pointers into the staging buffer, `read_offset` and `prefetch_offset` and we need to take care that the `prefetch_offset` never shoots ahead of the `read_offset` pointer, as this would imply that we are overwriting unconsumed data. `read_offset` is advanced whenever a new sample/batch is consumed and `prefetch_offset` is managed by the class. The buffer is

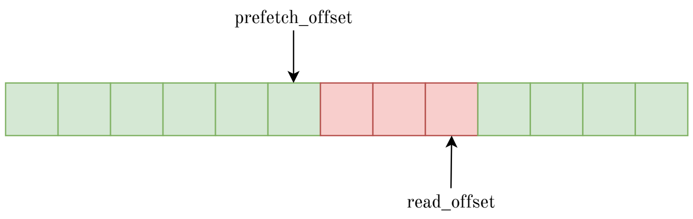

Figure 4.3: Staging Buffer Visualization.

illustrated in Figure 4.3, where green indicates blocks that still need to be consumed and therefore should not be overwritten whereas red blocks can be safely overwritten as the data was already consumed.

The `prefetch` method of the class is executed by the configured number of threads. When there is no work to do because the buffer is full, threads wait until they get notified by the consumer (i.e. the Python library). On the other hand, they also notify the consumer that is potentially waiting for new data to arrive when they fetched a sample/batch. As described earlier, we use a shared queue that contains the file ends to inform the consumer about samples/batches that can be consumed. We need to ensure that, although prefetching happens in parallel, the file ends are inserted into this queue in order, as decoding would fail otherwise. To do this, the threads first insert the file ends into a vector and push them onto the queue only if all previous ends were also pushed onto it. We use locks to ensure that different threads fetch different samples/batches, lay them out in a consecutive manner, and only one thread advances the epoch of the sampler. However, the actual fetching (which is generally the most time-consuming activity and also includes the application of transformations when they are configured) is done without holding a lock and can therefore be executed in parallel by multiple threads.

4.3.9 Prefetcher Backends

Prefetcher backends are the implementation of different storage class types. We provide the generic `PrefetcherBackend` interface which is implemented by concrete implementations. An implementation needs to provide:

- `prefetch()`: The method that is executed by the configured number of threads and does the actual prefetching. The method needs to call `insert_cached_file(storage_level, file_id)` of the provided metadata store as soon as it has finished fetching a sample.

- `fetch(file_id, *dst)`: When this method is called, the file with the given ID should be placed at the provided destination.

- `get_location(file_id, *len)`: Needs to return the memory address of the given file ID and the file length. The address is used by the distributed manager when sending files to other nodes. If a file is already in memory or addressable with a memory address (as in our two implementations), this address can directly be used to prevent additional copies. Otherwise, the backend needs to place the sample in memory and return the address.

- `get_prefetch_offset()`: Returns the current prefetch offset, used by the distributed manager for the availability heuristic.

- `is_done()`: Returns true if all the threads are done with prefetching and is also used by the distributed manager for the heuristic.

We provide two prefetcher backend implementations, `MemoryPrefetcher` for using RAM and `FileSystemPrefetcher` for prefetching to a file on an arbitrary storage device. `MemoryPrefetcher` allocates a buffer that is filled according to the prefetch string and stores the file ends in a vector. `fetch` is then implemented very easily, as we only need to get the sample location and issue a `memcpy`. `FileSystemPrefetcher` inherits from `MemoryPrefetcher` and reuses most of its logic. But instead of allocating a buffer, we map a file into memory with `mmap`. We use the node and job ID as the file name such that different jobs on the same node (or different MPI ranks on the same node) do not share the same file, which would lead to errors.

For the creation of the different backends, we make use of the abstract factory pattern, which allows easy extension of the framework with new backends. The only required change for the creation is to specify the name (as it will be used in the configuration file) and creation call in `PrefetcherBackendFactory`.

4.4 HDMLP - Data Augmentation

As pointed out earlier, HDMLP provides its own data transformations that can be optionally used which allows overlapping of data augmentation (including image decoding and conversion to target formats, e.g., tensors for PyTorch) with training and minimizes the number of data copies by storing transformed batches in memory.

4.4.1 Transformation Pipeline

For every transformation, there exists a corresponding Python class (in `hdmlp/lib/transforms`) that describes it (number and type of parameters, output dimensions based on input dimensions, output size in bytes based on input dimensions), but does not provide an implementation. As a list of these metadata objects is passed to the Python `Job` class, it can infer the size and dimensions of the pipeline's output. Note that variable sized inputs are supported, but the size of the output needs to be fixed (similar to other transformation pipelines like the one used by Torchvision). This output size and the name/arguments of the different transformations are passed to `libhdmlp` where they are used in the constructor of the `TransformPipeline` class. The class instantiates the different transformations and provides the `transform` method that takes a sample located in memory, applies all the steps of the pipeline sequentially, and places the transformed sample at a given location in memory. An implementation of a transformation in C++ implements the `Transformation` interface and therefore needs to provide the following two methods:

- `parse_arguments(char* arg_array)`: The type and number of arguments that are used when instantiating the corresponding Python class vary among different transformations. These are put into a memory buffer by the Python `Job` class which is passed as `arg_array` (with the correct offset, i.e. starting at the first argument of the transformation) to the transformation implementation. The implementation can then extract the arguments from this buffer and set its internal state accordingly. This design allows arbitrary-typed arguments with one uniform interface that is implemented by all transformations.
- `transform(TransformPipeline* pipeline)`: This is the actual implementation of the transformation. The function modifies all the fields of the pipeline that are needed to implement its functionality. Note that different transformations may act on different fields (for instance `ImgDecode` that accesses `src_buffer`, `src_len` and `img` whereas `Resize` only accesses `img`) and some operations can be performed in-place while others require a copy of some fields. To allow this flexibility with one uniform interface, we decided to provide the function with

a pointer to the pipeline instance. This also enables easy extension of the solution. If for instance the integration of a new machine learning framework requires a different output type, we would not need to change the interface but simply add this type as a field to the pipeline.

With this architecture, new transformations can be added very easily as they often only involve calls to other libraries (e.g., to OpenCV[6], which we use for image manipulations). For instance, our `Resize` implementation consists of only eight lines of code. At the moment, we provide the following transformations:

- `ImgDecode`: Decodes images in various file formats and should therefore be the first operations for transformations on images. As OpenCV's `imdecode` is used internally, all common file formats are supported.
- `Resize(w, h)`: Resizes the image to the desired size.
- `CenterCrop(w, h)`: Crops the image to the desired size at the center.
- `RandomVerticalFlip(p)/RandomHorizontalFlip(p)`: Flips images vertically/horizontally with probability p.
- `RandomResizedCrop(size, scale, ratio)`: The image is cropped where the size of the crop is a random multiple (with a range that is given by `scale`) of the original size and the aspect ratio is a random multiple (with its range determined by `ratio`) of the original ratio. Finally, the image is resized to `size`.
- `ToTensor`: Converts an image that is stored as an integer array (i.e. with integer values in the range $[0, 255]$) to a float array (i.e. with floating point values in the range $[0, 1]$), because the latter format is used by PyTorch's tensors for representing images.
- `Normalize(mean, scale)`: Normalizes an image, i.e. subtracts `mean` and divides by `scale`. Both parameters are 3-dimensional vectors where each dimension corresponds to a channel (R, G, and B) of the image.

[6]https://opencv.org/

Chapter 5

Evaluation

5.1 I/O Performance

For the formulation of our performance model (see Section 3.1), we ran several benchmarks to analyze the behavior of different storage types in scenarios with an access pattern similar to DNN training. The RAM and PFS benchmarks were run on Piz Daint XC50 compute nodes (see Section 5.3.1 for details) at the Swiss National Supercomputing Centre in Lugano, Switzerland. The SSD benchmark was run on a system with an Intel Core i7-8700B (6 hyperthreading-enabled cores), 32GB of RAM, and a NVMe SSD with a theoretical maximum sequential read bandwidth of 3.2GB/s. All tests were run ten times and we report the median value of the runs. Furthermore, we include the 95% nonparametric confidence interval, i.e. the value of the ranks j and k where

$$\mathrm{B}_{n,\frac{1}{2}}(k-1) - \mathrm{B}_{n,\frac{1}{2}}(j-1) \geq 1-\alpha \tag{5.1}$$

is satisfied for j and k [8]. With $n = 10$, $\alpha = 0.05$ and where $\mathrm{B}_{n,p}(x)$ denotes the cumulative distribution function of the binomial distribution with n repetitions and probability of success p, Equation 5.1 is satisfied for $j = 2$ and $k = 9$.

5.1.1 RAM

We used the STREAM [33] benchmark to measure the memory read bandwidth of an individual node. `gcc` 8.3.0 with the options `-fopenmp` and `-DSTREAM_ARRAY_SIZE=10000000` was used to compile the benchmark and we used the environment variable `OMP_NUM_THREADS` to control the number of threads. As we can see in Figure 5.1, the read bandwidth increases almost linearly until 9 threads. Using more than 9 threads does not result in a significant additional performance gain, which indicates that 9 threads can fully saturate the available memory bandwidth.

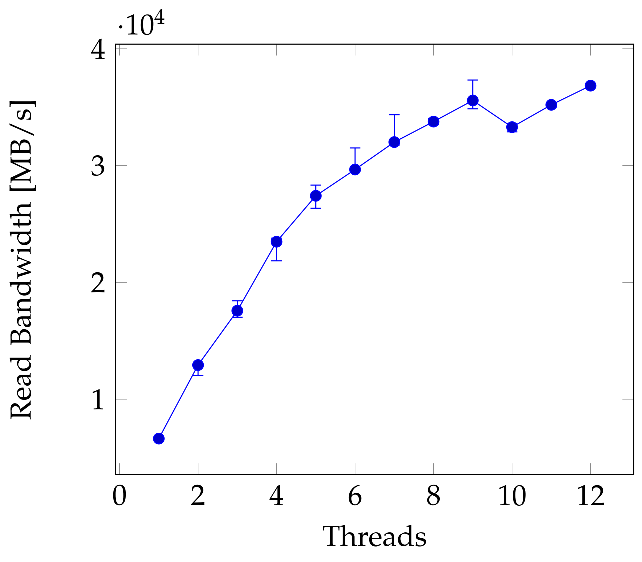

Figure 5.1: STREAM Benchmark Results.

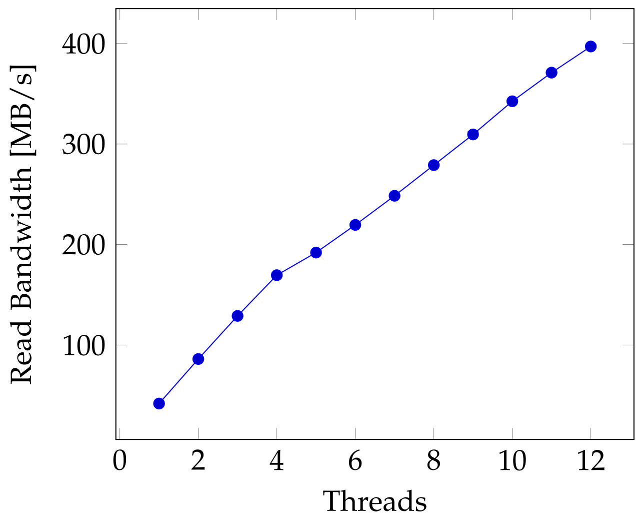

Figure 5.2: fio Benchmark Results.

5.1.2 SSD

To test the read bandwidth of the local storage, we used version 3.19 of the fio[1] benchmark. We set the block size (`bs`) to 4,096, simulated random reads (using `rw=randread`) and disabled buffered I/O (`direct=1`). `numjobs` was used to control the number of spawned threads and the total dataset size for each thread (`size`) was set to `4G`.

[1]https://fio.readthedocs.io/en/latest/fio_doc.html

As we can see in Figure 5.2, the read bandwidth scales almost linearly with the number of threads and the results are consistent among runs. The results suggest that even 12 threads are not able to fully saturate the read bandwidth (3.2GB/s) of the SSD in this particular setup.

5.1.3 PFS

For analyzing the behavior of the parallel file system with different number of reading nodes, we used the MPI-based IOR benchmark[2]. We set the parameters `blockSize` and `transferSize` to 4,096 which means that every rank contiguously wrote 4KB that were transferred in one go. `segmentCount` was set to 30,000, resulting in transfers of 30,000 segments (with size 4KB) per rank. To ensure that each MPI process is reading data it did not write (otherwise, the data could still be in the page cache, which would cause us to no longer measure the PFS performance), we set `reorderTasks` to 1. Furthermore, `randomOffset` was set to 1 for the simulation of random and not sequential access.

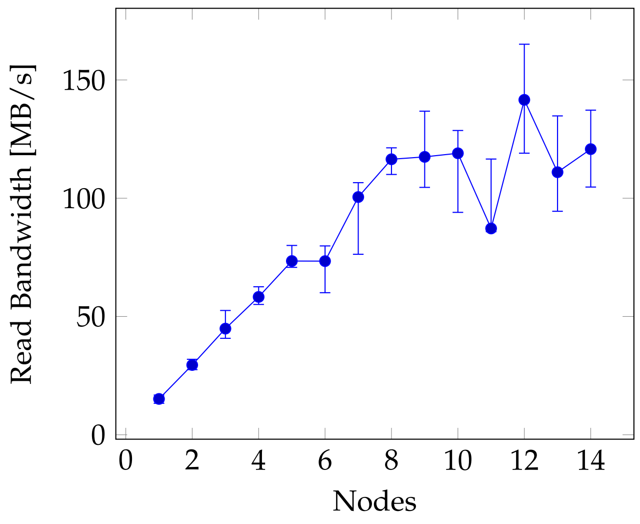

Figure 5.3: IOR Benchmark Results.

As we can see in Figure 5.3, the bandwidth grows almost linearly for up to 8 nodes and then fluctuates between 90MB/s and 140MB/s. It seems like 8 nodes are able to fully saturate the PFS in this particular setup as increasing the number of nodes beyond 8 did not result in additional performance benefits. Because the PFS is a shared resource, we observe wider confidence intervals than in the previous two benchmarks, especially for higher number of nodes.

[2] https://github.com/hpc/ior

5.2 Performance Simulation

We used our performance simulation to validate the design decisions we made in Section 3.2. For the sake of brevity, we report here only the results for one specific setup that simulates a small cluster. However, as pointed out in Section 4.1, the simulation can be used to simulate arbitrary environments and the behavior of different policies in these environments.

5.2.1 Methodology, Setup, Parameters

We simulated an environment consisting of 4 nodes with a compute throughput of 100MB/s. β (the preprocessing rate) was set to 200MB/s, b_c (the network bandwidth for inter-node communication) to 10,000MB/s, and b_{fs} (the bandwidth for communication with the PFS) to 2,000MB/s. We configured a 1,024MB staging buffer and 2 other storage levels. The first one is intended to simulate 50GB of RAM and the second one a local SSD with a capacity of 100GB. The number of prefetcher threads was set to 2 for every storage level and we simulated random read bandwidths $r_0(2) = 21{,}164\text{MB/s}$, $r_1(2) = 21{,}164\text{MB/s}$, and $r_2(2) = 86\text{MB/s}$. To model the PFS aggregate random read throughput, we set $t(1) = 66\text{MB/s}$, $t(2) = 86\text{MB/s}$, $t(3) = 129\text{MB/s}$, and $t(4) = 146\text{MB/s}$.

In the following scenarios, we assume normally distributed file sizes. We varied the parameters μ, σ, and number of samples F depending on the scenario, the chosen values are documented per scenario. In all scenarios, a global batch size of 100 was used. However, the number of epochs is different among scenarios.

For the HDMLP policy, we used the `FrequencyPrefetcher` class with `in_order` set to true as described in Section 4.1.4. We also investigated whether dropping files from storage classes after they are read for the last time results in an additional performance benefit, which can be activated by setting `cleanup_read` to true.

5.2.2 Results

We report the results from the four different scenarios we described in Section 4.1.

Scenario 1 - Data fits into node memory

We set $\mu = 0.027$, $\sigma = 0.01$, and $F = 10{,}000$. Therefore, the expected dataset size is 270MB, which is smaller than the RAM capacity.

The results of this setup are summarized in Table 5.1 (where the fastest policy is highlighted in bold) and Figure 5.4.

Policy	Runtime (s)
Lower Bound	7.30
Naive	27.78
HDMLP	7.78
StagingPoolPrefetcher	10.24
DeepIO (Ordered)	8.41
DeepIO (Opportunistic)	8.40
Parallel Data Staging	10.11
LBANN In Memory Store (Dynamic)	**7.71**
LBANN In Memory Store (Preloading)	8.41
Locality-Aware Data Loading	8.41

Table 5.1: Scenario 1 Results.

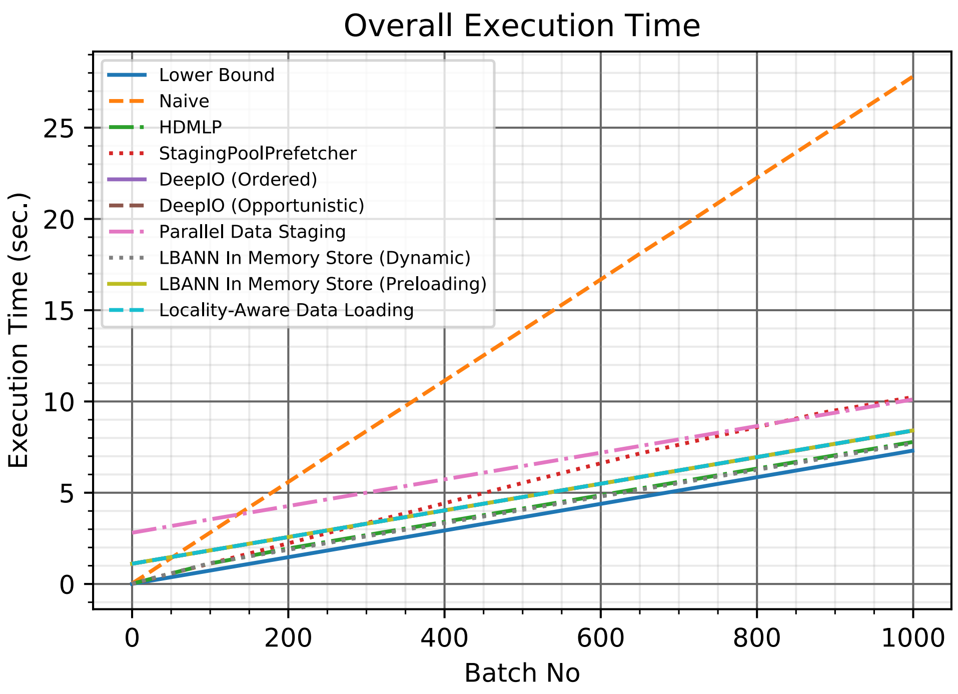

Figure 5.4: Scenario 1 Results.

As we can see, our policy performs well, although the differences are very small as this scenario is quite easy for all policies. But even in this scenario we can already see that proper handling of I/O is very important as the overall runtime of the naive policy is 3.6 times longer than the runtime of the best performing policy.

Scenario 2 - Data fits into node storage

In this scenario, we set $\mu = 1$, $\sigma = 0.1$, and $F = 150{,}000$ which gives an expected dataset size of 150GB. This no longer fits into the staging buffer or RAM alone but is still smaller than the aggregated storage of a single node.

Policy	Runtime (s)
Lower Bound	3,825.66
Naive	15,130.54
HDMLP	**3,836.12**
StagingPoolPrefetcher	4,982.27
DeepIO (Ordered)	4,433.27
DeepIO (Opportunistic)	4,432.90
Parallel Data Staging	4,972.65
LBANN In Memory Store (Dynamic)	4,051.12
LBANN In Memory Store (Preloading)	4,433.27
Locality-Aware Data Loading	4,433.27

Table 5.2: Scenario 2 Results.

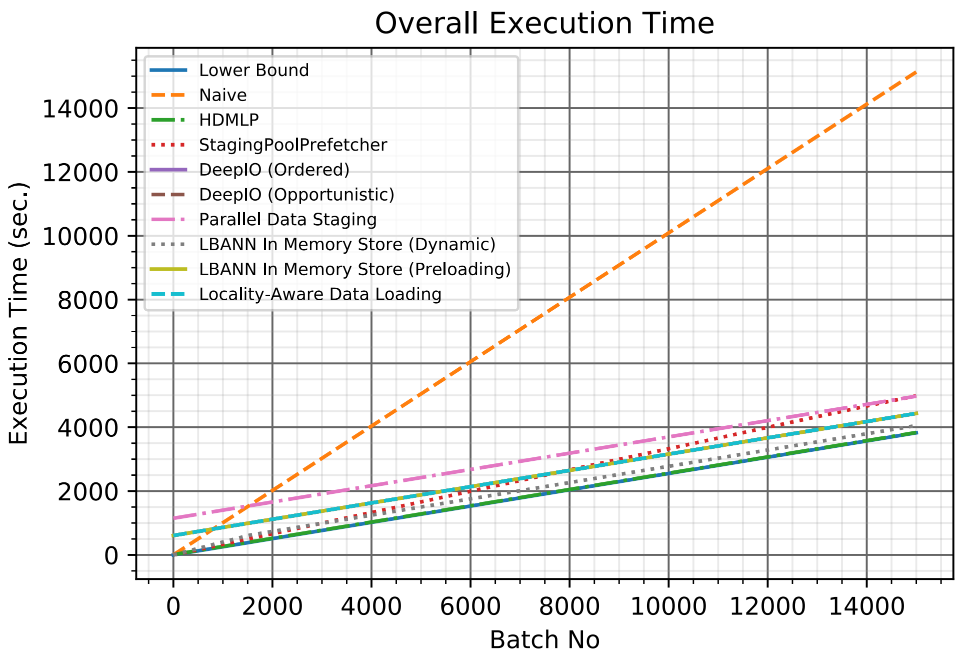

Figure 5.5: Scenario 2 Results.

As we can see in Table 5.2 and Figure 5.5, our policy performs very well in this scenario too and is close to the theoretical lower bound. Our analysis showed that HDMLP performed better than other policies because it does not require an initialization phase (in contrast to policies like parallel data staging), reduces the number of PFS reads (whereas other policies like StagingPoolPrefetcher always read from the PFS), and utilizes all available resources (compared to policies like LBANN's In Memory Store or DeepIO that only use RAM which is too small for storing the whole dataset in this setup).

Scenario 3 - Data fits into aggregated node memory

Here, we chose $\mu = 2$, $\sigma = 0.2$, and $F = 300{,}000$, resulting in an expected dataset size of 600GB which is more than the aggregated storage of a single node but less than the aggregated storage of all nodes.

Policy	Runtime (s)
Lower Bound	7,655.25
Naive	30,272.96
HDMLP	**7,661.26**
StagingPoolPrefetcher	9,945.29
DeepIO (Ordered)	23,948.24
DeepIO (Opportunistic)	8,487.42
Parallel Data Staging	9,313.58
LBANN In Memory Store (Dynamic)	N/A
LBANN In Memory Store (Preloading)	N/A
Locality-Aware Data Loading	9,314.33

Table 5.3: Scenario 3 Results.

As in the previous two scenarios, we almost hit the lower bound, i.e. the stall time with HDMLP is almost zero. Note that LBANN's in-memory store does not support datasets that are larger than the aggregated RAM size, therefore we do not have results for scenarios 3 and 4. We can also see that DeepIO in ordered mode performs poorly for setups where the dataset is much larger than the aggregated RAM size. This is due to the fact that many samples need to be fetched from the PFS, especially because they do not consider the access frequency for the prefetching decision which makes it possible that a node with a high access frequency for a given sample always reads it from the PFS. Their solution to this problem (entropy-aware opportunistic ordering) improves performance at the expense of accuracy.

Scenario 4 - Data does not fit into aggregated node memory

To simulate the fourth case, we set $\mu = 3$, $\sigma = 0.2$, and $F = 400{,}000$ which means that the expected dataset size is 1,200GB and therefore larger than the aggregated node storage. As we can see in Table 5.4 and Figure 5.7, our policy performs very well without changing the access string even in this challenging setup. The only policy that performs better in this scenario

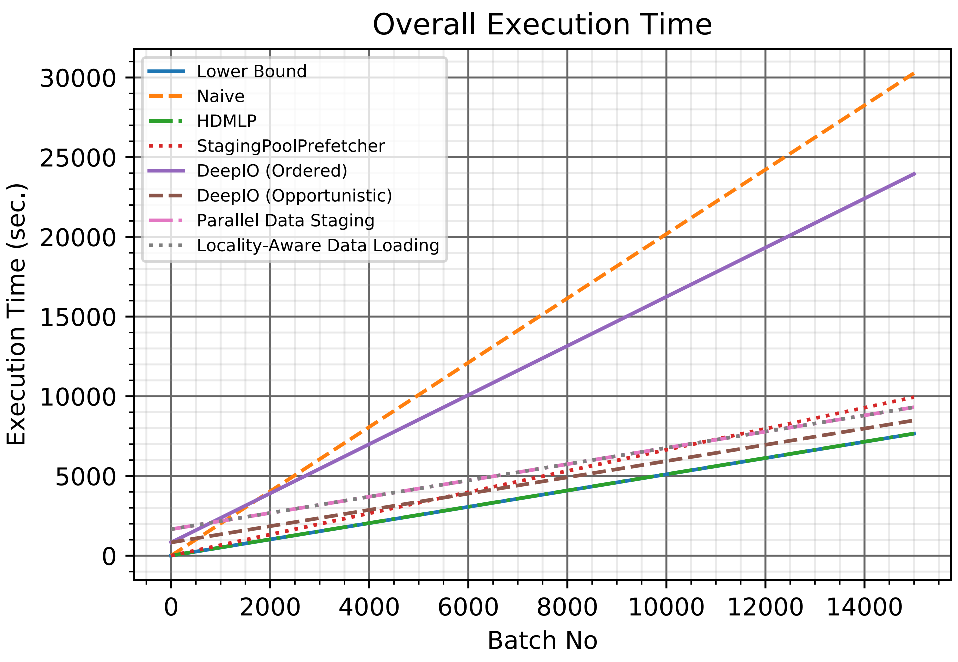

Figure 5.6: Scenario 3 Results.

Policy	Runtime (s)
Lower Bound	15,204.78
Naive	60,526.36
HDMLP	16,795.17
StagingPoolPrefetcher	19,888.26
DeepIO (Ordered)	54,122.52
DeepIO (Opportunistic)	**16,034.89**
Parallel Data Staging	16,861.82
LBANN In Memory Store (Dynamic)	N/A
LBANN In Memory Store (Preloading)	N/A
Locality-Aware Data Loading	19,082.99

Table 5.4: Scenario 4 Results.

is DeepIO in opportunistic mode. The policy can benefit from the fact that it only considers available samples at any given time, therefore the large size of the dataset does not influence its runtime as much. Interestingly, Parallel Data Staging outperforms Locality-Aware Data Loading although it requires an initialization phase. But because the policy uses node-local storage, the overall runtime for this rather long run is still shorter than the one of Locality-Aware Data Loading which only uses RAM.

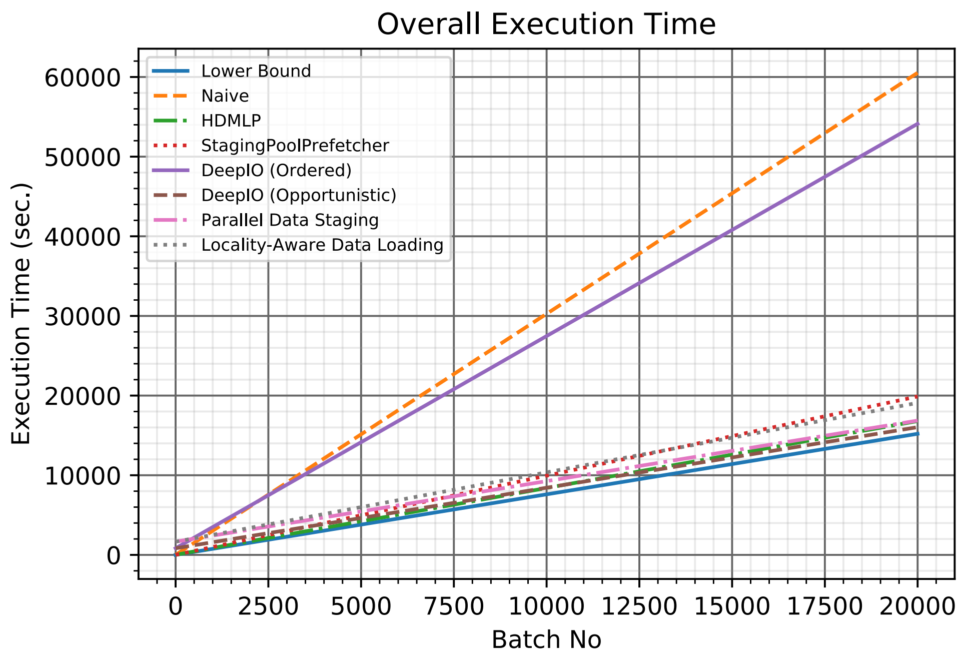

Figure 5.7: Scenario 4 Results.

5.2.3 Environment Evaluation

As pointed out in Section 4.1, the performance simulation can also be used to quantify the impact of changes to the environment on the runtime of neural network training. To illustrate this, we consider Scenario 3 from Section 5.2.2 again. We set the policy to HDMLP and ran the simulation in different system configurations. To get a baseline value, we first simulated a configuration where no local storage is available (meaning the samples are fetched from the PFS when they are needed without any buffering) and got an execution time of 30,275.72 seconds. We also used the simulation to get the lower bound, which is 7,655.47 seconds for this particular scenario. We then simulated setups with a staging buffer of size 1GB, 2GB, and 4GB which all resulted in runtimes of 9,947 seconds (indicating that the size of the staging buffer is not a limiting factor in this configuration). For the following runs, we kept the staging buffer size fixed at 4GB.

In the next experimental runs, we added 10GB, 20GB, and 40GB of RAM as an additional storage class. As we can read off from Table 5.5, using more memory for buffering speeds up training, but we are not able to reach the lower bound with 40GB per node.

To see whether using RAM for the buffer is necessary or we can get similar speedups with other, cheaper types of storage, we simulated a setup

Configuration	Runtime (s)
10GB RAM	9,554.47
20GB RAM	9,239.67
40GB RAM	8,613.13

Table 5.5: Evaluation with Different In-Memory Buffer Sizes.

where each node is equipped with one additional storage class (in addition to the staging buffer). The random read bandwidth of this class was set to 100MB/s for 2 threads (i.e. $r_1(2) = 100\text{MB/s}$). We ran the experiment with 20GB, 40GB, and 80GB per node. The results of these runs are documented in Table 5.6. Interestingly, the runtimes when using 80GB of this type of storage are very similar to the setup with 40GB RAM. Therefore, investing in more memory may not be worthwhile if such a scenario is representative for an environment.

Configuration	Runtime (s)
20GB storage	9,579.53
40GB storage	9,257.99
80GB storage	8,703.59

Table 5.6: Evaluation with Different Storage Device Buffer Sizes.

We also simulated configurations where nodes are equipped with different combinations of RAM and storage of the type mentioned above. The results of these simulations are illustrated in Figure 5.8.

5.3 Training

5.3.1 Methodology, Setup, Parameters

Experimental Setup and Architectures

The CSCS Piz Daint supercomputer was used for the following benchmarks. Every XC50 compute node of the system is equipped with a 12-core hyper-threading-enabled Intel Xeon E5-2690 CPU, 64GB RAM, and one NVIDIA Tesla P100 GPU with 16GB memory. Cray's Aries interconnect with a 36TB/s bisection bandwidth is used to connect the nodes.

The datasets are initially located on a shared Lustre filesystem. The tests were performed sequentially to minimize the potential influence of PFS caching on the results. However, as the PFS is a shared resource, a potential influence by other workloads running in parallel is unavoidable. We believe that these influences should not be significant for the results, which we con-

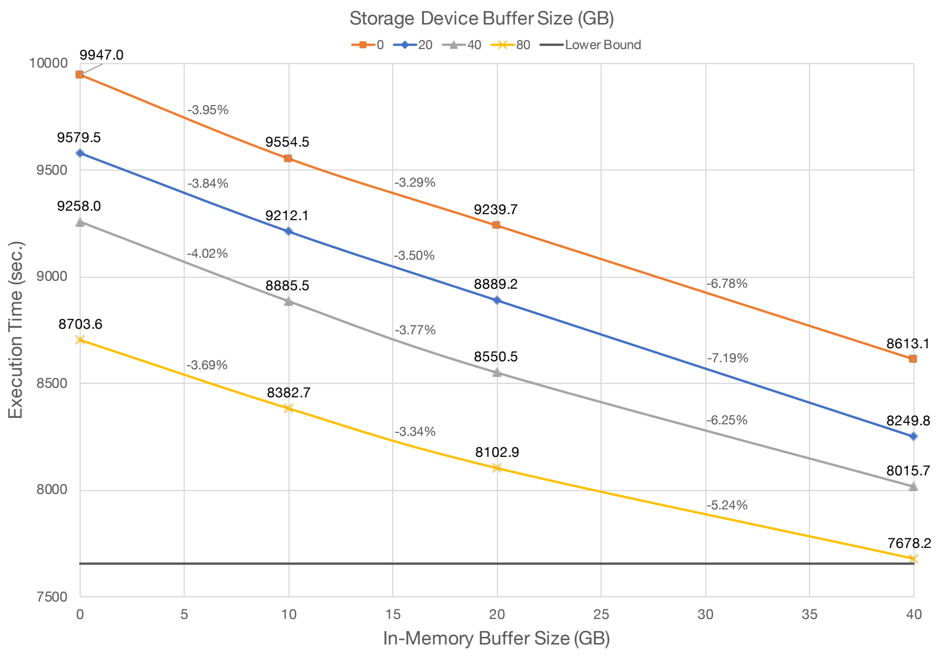

Figure 5.8: Environment Evaluation Results.

firmed by running some tests repeatedly at different times and comparing the results.

Evaluation Methodology

In the first two benchmarks, we measured the raw framework performance without any additional influences. For this, we wrote a benchmark script that simulates distributed training of a neural network but simply iterates over the data without doing anything with it. This is a particularly challenging setup as it simulates a scenario where new samples are consumed immediately. We measured and report the average load time among all nodes. We compare our `HDMLPImageFolder` implementation against PyTorch's `ImageFolder`. In the first two experiments, we configured HDMLP to use a staging buffer of size 2,048MB and additional 4,096MB of RAM for prefetching. Furthermore, we configured the framework to use two threads each for prefetching to the staging buffer and RAM.

To ensure that the framework also works in practical deployments, we used PyTorch to implement a distributed image classifier and trained different neural networks in different scenarios. We compared the validation accuracy and ensured that there are no significant differences among the data loaders. In this benchmark, we measured the stall time (the time each node spent waiting for the transformed data to arrive) and report the median value among all the nodes. We configured HDMLP to use its internal data transformations and compare it against:

- **DataLoader:** In this setup, PyTorch's `DataLoader`[3] is used for the data loading. The class is used in the single-process data loading mode, meaning `num_workers` is set to 0 and data loading is done by the main process. We pass a `DistributedSampler` to the data loader to ensure that the samples are distributed among the nodes.
- **Multi-Threaded Prefetching:** As in the previous case, we use the `DataLoader` class with a `DistributedSampler`. However, we set `num_workers` to 4 which causes the data loader to prefetch data with 4 processes.

In this experiment, we set the size of HDMLP's staging buffer to 4,096MB and used 20,480MB of RAM as one additional storage class. Four threads were configured for prefetching to the staging buffer, two for the RAM.

Because HDMLP requires the thread support level `MPI_THREAD_MULTIPLE`, we had to set `MPICH_MAX_THREAD_SAFETY` to `multiple` on Piz Daint. For this reason, our implementation ran with a slower/non-optimized MPI library, potentially worsening the results.

We used the following software versions in our benchmarks:

- Python 3.7.4
- PyTorch 1.4.0a0+7404463 (compiled from source including the NCCL backend)
- TorchVision 0.5.0
- OpenCV 4.3.0
- NCCL 2.6.4
- cuDNN 7.6.5
- CUDA 10.1.105
- Cray MPICH 7.7.10

gcc 8.3.0 was used to compile the software.

5.3.2 Experiment 1: Hymenoptera Dataset

We simulated training on a small ImageNet subset, containing images of ants and bees. The dataset consists of 245 images with a total size of 27MB. In our simulation, we set the number of epochs to 100.

[3] https://pytorch.org/docs/stable/data.html

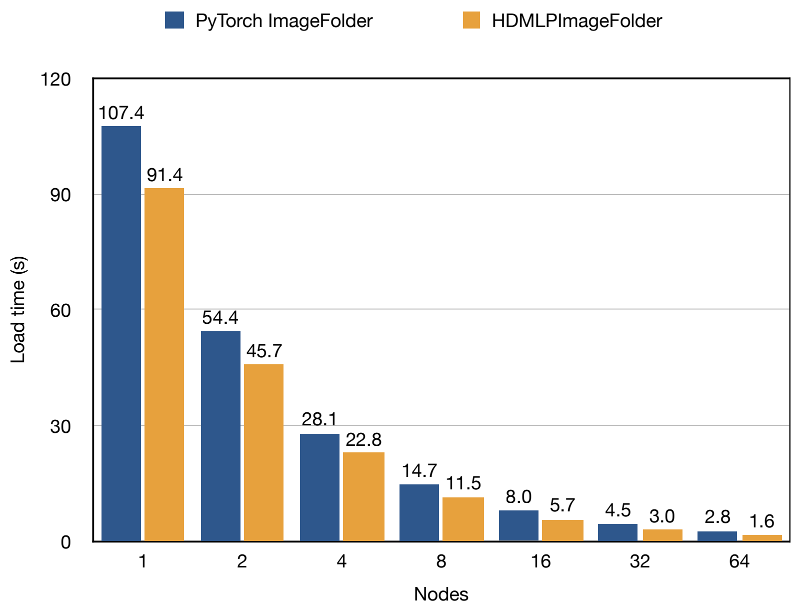

Figure 5.9: Hymenoptera Image Loading Benchmark Results.

	PyTorch ImageFolder		HDMLPImageFolder	
Nodes	**Strong Scaling Speedup**	**Strong Scaling Efficiency**	**Strong Scaling Speedup**	**Strong Scaling Efficiency**
2	1.973	0.986	1.998	0.999
4	3.827	0.957	4.008	1.002
8	7.301	0.913	7.960	0.995
16	13.401	0.838	15.9	0.994
32	23.872	0.746	30.952	0.967
64	38.946	0.609	56.271	0.879

Table 5.7: Strong Scaling Speedup and Efficiency of HDMLP.

The results of the experiment are depicted in Figure 5.9. For every configuration of number of nodes, HDMLP performed better than PyTorch's `ImageFolder` class. In Table 5.7, we compare the strong scaling speedup ($S_p = \frac{t_1}{t_p}$, where t_1 is the load time of one node and t_p that of p nodes) and efficiency ($\frac{S_p}{p}$) of our implementation and PyTorch's `ImageFolder`. As we can see in the table, HDMLP scales well to many nodes, achieving strong scaling efficiencies of almost 1 for up to 32 nodes. The scaling behavior is much better than the one we observed with PyTorch's `ImageFolder`. For instance, with 32 nodes we achieve a strong scaling speedup of 30.95 whereas the speedup of the `ImageFolder` class is only 23.87. This indicates that our framework indeed scales to many nodes with minimal overhead.

5.3.3 Experiment 2: ImageNet Dataset

In the second experiment, we used a much larger subset of the ImageNet dataset. It consists of 300,000 images in 1,000 different classes with a total size of 33GB. We simulated training of a neural network for one epoch.

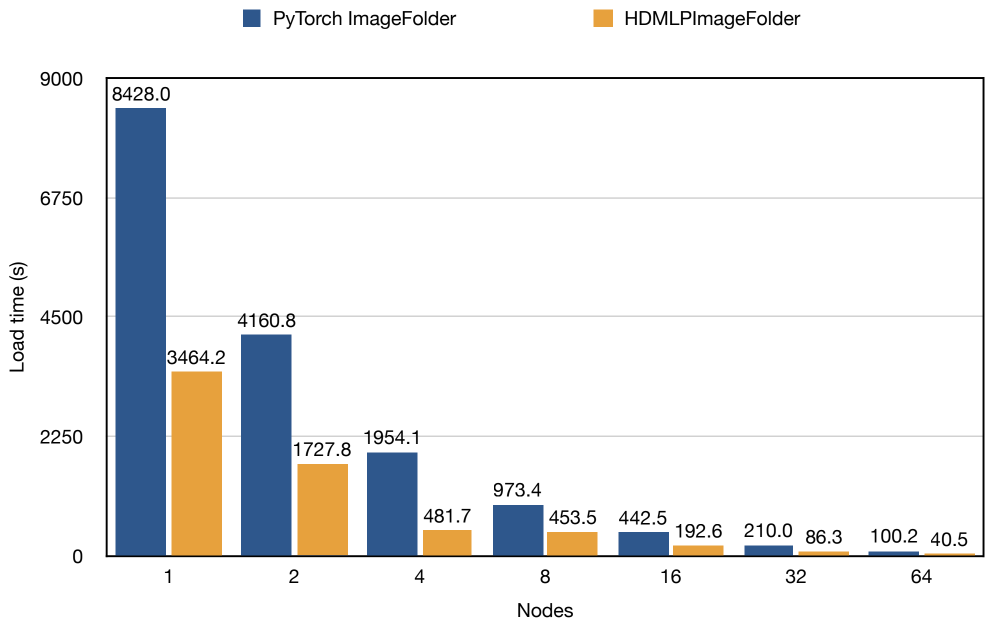

Figure 5.10: ImageNet Image Loading Benchmark Results.

Nodes	Speedup	Strong Scaling Speedup	Strong Scaling Efficiency
1	2.433	1	1
2	2.408	2.005	1.002
4	4.057	7.192	1.798
8	2.147	7.640	0.955
16	2.298	17.987	1.124
32	2.433	40.146	1.255
64	2.476	85.591	1.337

Table 5.8: Speedup, Strong Scaling Speedup, and Efficiency of HDMLP.

As we can see in Figure 5.10, HDMLP performed much better than PyTorch's `ImageFolder`, achieving speedups of up to 4x. In Table 5.8, we report the achieved speedup values (compared to `ImageFolder`), the strong scaling speedup, and efficiency. As in the previous experiment, we observe very good strong scaling efficiency with values greater than 1 for almost all runs.

5.3.4 Experiment 3: Image Classification

In the following experiments, we trained different neural network architectures on different-sized datasets to see how HDMLP performs in a practical situation. In all benchmarks, a node-local batch size of 128 was used (meaning we scaled the global batch size by the number of nodes). Before feeding the data into the model, the following transformations were applied:

- **Training:**
  - Randomly cropping and resizing the image (`RandomResizedCrop`)
  - Randomly flipping the image horizontally with a probability of 0.5 (`RandomHorizontalFlip`)
  - Converting the image to a tensor (`ToTensor`)
  - Normalizing the tensor image (`Normalize`)
- **Validation:**
  - Resizing the image (`Resize`)
  - Cropping the image at the center (`CenterCrop`)
  - Converting the image to a tensor (`ToTensor`)
  - Normalizing the tensor image (`Normalize`)

Benchmark 1: ResNet-50 on 10% of ImageNet

In this benchmark, we trained ResNet-50 [19] on a randomly sampled subset of ImageNet with 100 images per class (therefore 100,000 in total) for 3 epochs. The total dataset size was 11GB, the benchmark therefore corresponds to Scenario 1 from Section 4.1. As we can see in Figure 5.11, HDMLP significantly reduced the median stall by a factor of up to 1,132 compared against PyTorch's data loader and up to 36 compared against the multithreaded prefetcher. Starting from four nodes, we are able to completely overlap I/O and computation as the median stall time for the whole training run is very low.

Benchmark 2: SqueezeNet on 10% of ImageNet

Here, we trained the SqueezeNet [20] model on the same dataset as in the previous benchmark for 3 epochs. We have chosen this model because it is small and therefore allows us to test the data loading mechanism in a demanding setup where the compute throughput is high. The results of this benchmark are illustrated in Figure 5.12. As expected, the stall time is much larger for all implementations because samples are consumed faster. However, HDMLP still performs very well and has a lower median stall time in all runs.

Benchmark 3: ResNet-50 on ImageNet

In this experiment, we used 8, 16, and 32 nodes for training ResNet-50 on the whole ImageNet dataset, i.e. 1,281,167 samples in 1,000 classes with a

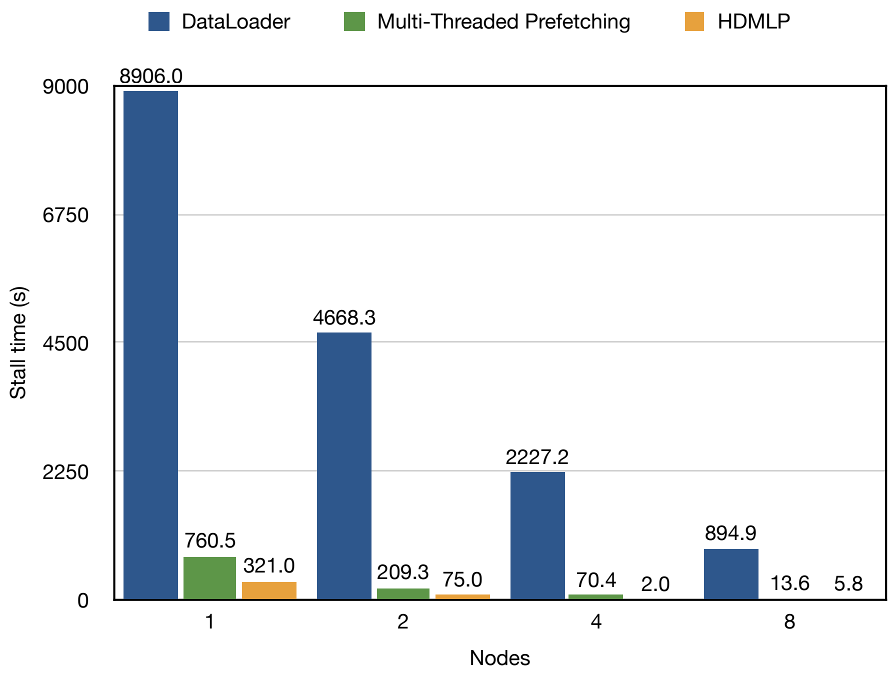

Figure 5.11: ResNet-50 ImageNet (10%) Classification Benchmark Results.

total dataset size of 138GB. Training was performed for 3 epochs and as before, we report the median stall time in Figure 5.13. As we see from the results, HDMLP was able to completely overlap I/O and computation in all runs. The usage of the framework reduced the median stall time by factors of up to 2,924 and 44 compared to PyTorch's data loader and multi-threaded prefetcher.

Benchmark 4: Large-Scale ImageNet Training

Like in the previous benchmark, we trained ResNet-50 on the whole ImageNet dataset. To see how well the framework performs when many nodes are used, we ran the experiment with 256 nodes. This setup is especially interesting because we observed severe degradation of the PFS performance when using this many nodes for distributed training on Piz Daint in the past. We performed the benchmark two times for 10 and 60 epochs respectively. In Figure 5.14, we report the maximum stall time (per iteration) among nodes. In spite of the PFS performance, we were able to completely overlap I/O and computation in the first run where training was performed for 10 epochs. It seems to be beneficial that HDMLP only considers the samples that a node will access for prefetching and requests them from the PFS according to the first reference. In the second run, there are phases where complete overlap is achieved but this is not always the case, especially not in the beginning. A closer analysis of the results showed that for the slowest node, not enough

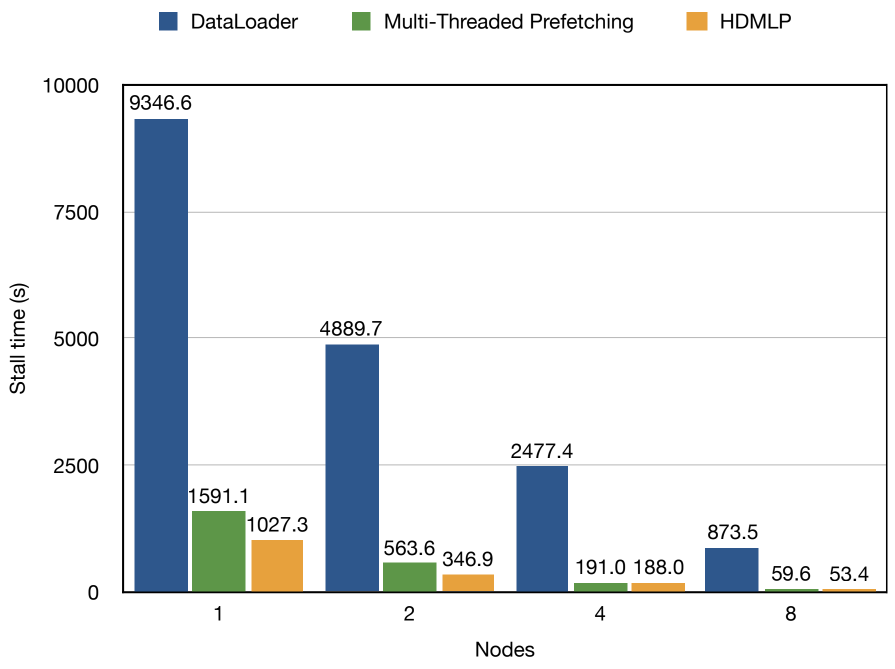

Figure 5.12: SqueezeNet ImageNet (10%) Classification Benchmark Results.

samples were cached (as the in-memory buffer was filled slowly because of the PFS performance) and some were still fetched from the PFS, which caused these stall times.

5.4 Discussion

In Section 2.3, we stated five properties that an ideal system should have and concluded that the existing solutions fail to meet all of these requirements at the same time. We now want to discuss why HDMLP satisfies all of these properties:

1. *Dataset Scalability*: We do not place any restrictions on the dataset size. The dataset can be larger than the aggregated node storage, but we also support small datasets. In our benchmarks, we have shown that the framework performs well for different datasets of very different sizes.

2. *Node Scalability*: Support for many nodes was a key concern when designing the solution, which among others resulted in our distributed caching approach with no additional metadata messages and very little overhead. In our experiments, we confirmed that our framework indeed scales very well, outperforming the strong scaling efficiency of PyTorch's `ImageFolder` with values close to 100% and achieving sig-

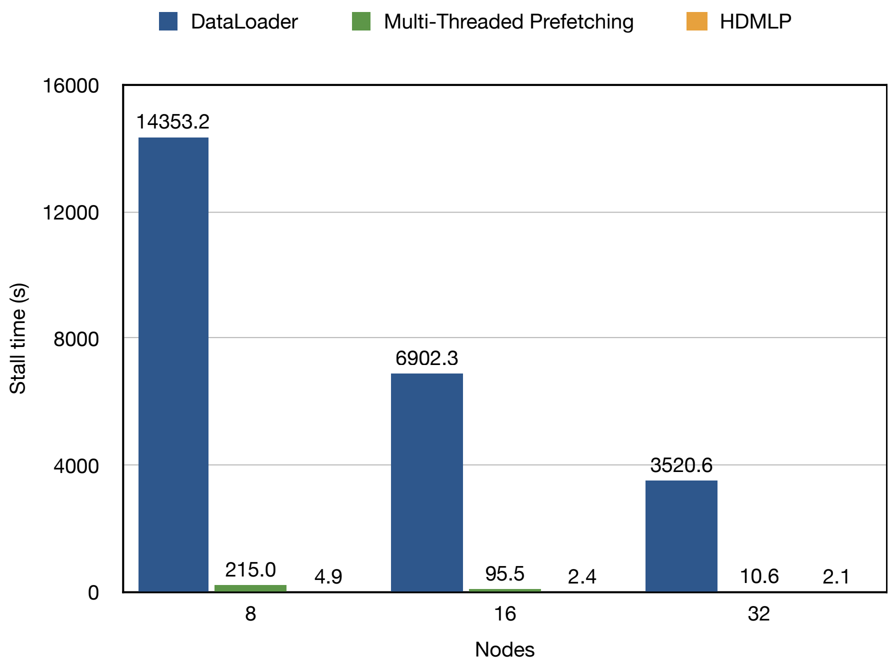

Figure 5.13: ResNet-50 ImageNet Classification Benchmark Results.

nificant speedup values in realistic benchmarks.

3. *Configuration Independence*: As described in Section 4.3, our system is completely configurable and we assume very little about the environment. Besides configurability, the solution is built in a modular way so that a user can easily extend it with new distribution schemes, transformations, or prefetcher and storage backends, therefore providing even greater flexibility.

4. *Model Accuracy*: We completely shuffle the whole dataset every epoch and do not change the mini-batch sequences in any way. Therefore, our framework does not impact the model accuracy.

5. *Ease of Use*: We have shown in Section 4.2 that our PyTorch implementation requires the user to only change 3 lines of code, otherwise he can continue to use his existing codebase. HDMLP is therefore very easy to integrate into existing codebases. Furthermore, the datasets and data loaders can be kept simple and short because many features that otherwise would be implemented there are already provided by the library, especially if the integrated transformations are used.

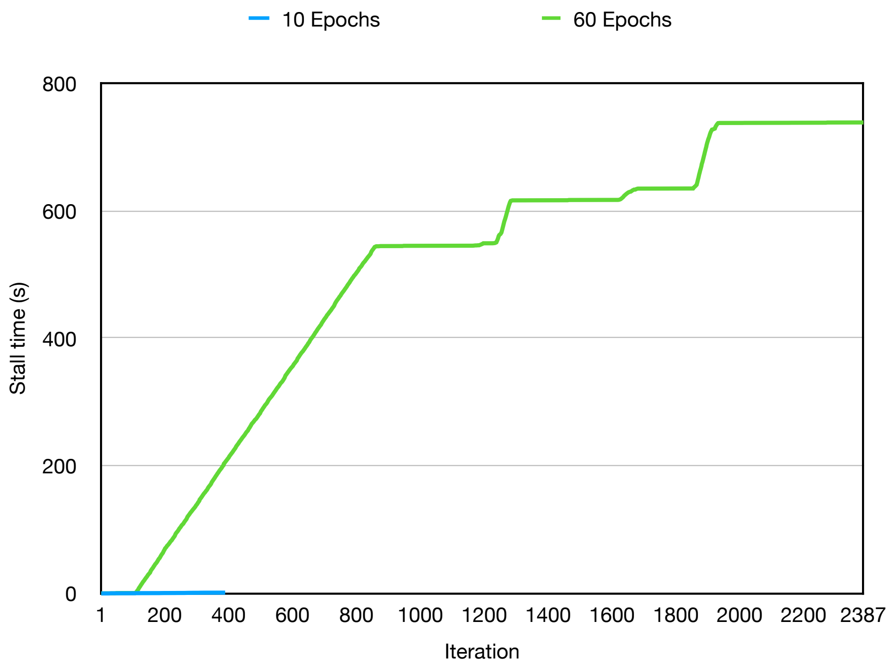

Figure 5.14: Large-Scale ImageNet Training Results.

Chapter 6

Related Work

We have already presented and analyzed several existing solutions for machine learning data loading, caching, and prefetching in Section 2.3. As we pointed out there, none of the existing approaches meets all the desired criteria. HDMLP is the first solution that provides dataset & node scalability, configuration independence, accuracy, and ease of use at the same time.

The problem of caching and prefetching has been studied for many years in different contexts. In 1966, Bélady studied different page replacement algorithms and proved the optimality of Bélady's MIN algorithm [4]. In 1995, Cao et al. studied prefetching and caching in an integrated fashion and derived four properties that optimal integrated strategies must satisfy, while also presenting two strategies that satisfy all of these properties [9]. In 2003, Ambühl and Weber proved the NP-hardness of the parallel prefetching and caching problem [3]. Distributed and hierarchical caching has been studied in various contexts, among others for video-on-demand content [26] and content delivery networks [7]. We incorporated several results of these analyses into our design.

A lot of research has been conducted in the area of parallel and distributed deep learning. The survey paper by Ben-Nun and Hoefler [6] discusses and analyzes the different approaches and parallelization strategies that are used today. The survey was used as a foundation to determine the target environments and their properties. Some prior work also focuses on the analysis of certain subsystems that are involved with I/O in the context of machine learning: Chow et al. analyzed the behavior of the parallel file system implementation BeeGFS under deep learning workloads [13] and Pumma et al. analyzed the deep learning I/O subsystem LMDB and proposed several optimizations for it [37]. Chien et al. characterized the I/O performance and scaling of TensorFlow [12], especially the use of its prefetcher and burst buffer. Different insights of those works were used during the design process of HDMLP.

Chapter 7

Further Work

The problem of data loading in the context of machine learning will become even more important in the future, due to growing dataset sizes and further computational improvements. We believe that HDMLP provides a great foundation to become a widely used and deployed machine learning I/O middleware. For that, it would be important to integrate additional machine learning frameworks such that they can use HDMLP with minor code changes (similarly to our PyTorch datasets/data loader). Because flexibility, configurability, and modularity were key concerns during design and development, many extensions and additional features can be added to the framework. As we have already pointed out in Section 4.3.4, one possibility for further development would be to implement specialized storage backends that for instance support the HDF5 file format or can access object stores over specialized APIs. Depending on the use case, other prefetcher backends could also provide additional benefits and could be implemented very easily, as described in Section 4.3.9. For instance, existing key-value stores could be integrated into our framework by implementing the required access methods.

As an extension to our `StagingBufferPrefetcher` class, it would be interesting to explore the potential benefits of using pinned memory and prefetching directly to a staging buffer on the GPU, for instance using CUDA's specialized memory management APIs[1], similar to TensorFlow's experimental `prefetch_to_device`[2]. In some scenarios, copying data to the GPU memory can become a bottleneck which could be eliminated like this.

In our performance model, we assumed that the PFS read bandwidth per number of clients is fixed. However, there are differences between sequential and random accesses in practice. It would be interesting to explore how

[1] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html

[2] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/data/experimental/ops/prefetching_ops.py

our model can be extended to incorporate these characteristics and use it to analyze the tradeoff between reading consecutive data, according to the current scheme (by the access string for the staging buffer and the access frequency for the other storage levels), or some hybrid scheme (e.g. determining ranges of consecutive samples that a node will access often times or incorporating data sieving and collective I/O similar to ROMIO [41]). If we drop the assumption that we are only reading data, we could further explore in which scenarios an initial reordering of the data on the PFS (to improve the access pattern during training) decreases overall stall time and in which way the samples should be laid out ideally. It would also be interesting to analyze the potential benefits of dynamic migrations (that exchange samples with other local storage classes/nodes during training or drop them) on the performance, both from a theoretical and practical perspective.

Currently, the user has to specify environment parameters like the PFS bandwidth or number of threads in a configuration file. It would be interesting to explore if these parameters could be inferred and set (e.g. managing the number of threads by measuring the improvements of using additional threads) automatically during execution. Furthermore, there are setups where multiple processes are spawned per node (e.g. for nodes that are equipped with multiple GPUs). Incorporating this fact into our performance model and design (for instance by preferring ranks on the same node and using shared memory to communicate with them) could result in additional performance gains in such setups.

Besides machine learning, our solution could also be applied with very little modifications to other types of problems that exhibit similar characteristics, meaning the access pattern is known a priori or can be inferred easily and there are many small, noncontiguous reads. For instance, it has been observed that applications in the life-sciences industry such as genome sequencing exhibit this type of access pattern (because intermediate sequencing results are often stored noncontiguously) and therefore can saturate the parallel file system if I/O is not handled properly [27,30].

Chapter 8

Conclusion

I/O can be a major bottleneck during the training of neural networks (especially in distributed deployments) and there is no satisfactory solution that addresses this issue. This research aimed to analyze the problem of data loading, caching, and prefetching in the context of machine learning and develop a new solution in accordance with the analysis. Based on the survey of current approaches and their shortcomings, an analytical performance model, and problem-specific insights, we derived a novel design for a machine learning I/O middleware that minimizes the stall time during training. Our design makes use of stochastic gradient descent's random nature which results in different sample access frequencies among nodes. It maximizes the usage of the available resources by a model-driven fetching decision as well as hierarchical, distributed caching and prefetching. With HDMLP, we provide a performant, extensible, flexible, and configurable implementation of this design that can be easily used by machine learning practitioners. In our evaluation we show that HDMLP satisfies the five core challenges we identified: Dataset and node scalability, configuration independence, model accuracy, and ease of use. In practical benchmarks that simulate realistic scenarios, we demonstrate that our solution can reduce the stall time by factors of up to 2,924 and therefore significantly reduces overall training time, enabling much higher throughputs.

We believe that the proper handling of I/O in the context of machine learning will become even more important in the future and HDMLP provides the foundation to solve this problem once and for all.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Savannah, GA, USA, November 2016. USENIX Association.

[2] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. YouTube-8M: A Large-Scale Video Classification Benchmark. *arXiv:1609.08675 [cs]*, September 2016.

[3] Christoph Ambühl and Brigitta Weber. Parallel prefetching and caching is NP-hard. Report, ETH Zurich, 2003.

[4] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[5] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 66–77, Rio de Janeiro, Brazil, May 2019. IEEE.

[6] Tal Ben-Nun and Torsten Hoefler. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Computing Surveys*, 52(4):65:1–65:43, August 2019.

[7] Sem Borst, Varun Gupta, and Anwar Walid. Distributed Caching Algorithms for Content Distribution Networks. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, March 2010.

[8] Jean-Yves Le Boudec. *Performance Evaluation of Computer and Communication Systems*. EPFL Press, 2011.

[9] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, 23(1):188–197, May 1995.

[10] Chen Chen, Qizhen Weng, Wei Wang, Baochun Li, and Bo Li. Fast Distributed Deep Learning via Worker-adaptive Batch Sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 521, Carlsbad, CA, USA, October 2018. Association for Computing Machinery.

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *arXiv:1512.01274 [cs]*, December 2015.

[12] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. Characterizing Deep-Learning I/O Workloads in TensorFlow. *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 54–63, November 2018.

[13] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/O Characterization and Performance Evaluation of BeeGFS for Deep Learning. In *Proceedings of the 48th International Conference on Parallel Processing - ICPP 2019*, pages 1–10, Kyoto, Japan, 2019. ACM Press.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.

[15] Richard Galvez, David F. Fouhey, Meng Jin, Alexandre Szenicer, Andrés Muñoz-Jaramillo, Mark C. M. Cheung, Paul J. Wright, Monica G. Bobra, Yang Liu, James Mason, and Rajat Thomas. A Machine-learning Data Set Prepared from the NASA *Solar Dynamics Observatory* Mission. *The Astrophysical Journal Supplement Series*, 242(1):7, May 2019.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[17] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv:1706.02677 [cs]*, April 2018.

[18] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A Survey of Deep Learning Techniques for Autonomous Driving. *Journal of Field Robotics*, 37(3):362–386, April 2020.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.

[20] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs]*, November 2016.

[21] Sam Ade Jacobs, Brian Van Essen, David Hysom, Jae-Seung Yeom, Tim Moon, Rushil Anirudh, Jayaraman J. Thiagaranjan, Shusen Liu, Peer-Timo Bremer, Jim Gaffney, Tom Benson, Peter Robinson, Luc Peterson, and Brian Spears. Parallelizing Training of Deep Generative Models on Massive Scientific Datasets. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–10, September 2019.

[22] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes. *arXiv:1807.11205 [cs, stat]*, July 2018.

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, pages 675–678, Orlando, Florida, USA, November 2014. Association for Computing Machinery.

[24] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben

Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *ACM SIGARCH Computer Architecture News*, 45(2):1–12, June 2017.

[25] Harsh Khetawat, Christopher Zimmer, Frank Mueller, Scott Atchley, Sudharshan S. Vazhkudai, and Misbah Mubarak. Evaluating Burst Buffer Placement in HPC Systems. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, Albuquerque, NM, USA, September 2019. IEEE.

[26] Christian Koch, Johannes Pfannmüller, Amr Rizk, David Hausheer, and Ralf Steinmetz. Category-aware hierarchical caching for video-on-demand content on youtube. In *Proceedings of the 9th ACM Multimedia Systems Conference*, MMSys ’18, pages 89–100, Amsterdam, Netherlands, June 2018. Association for Computing Machinery.

[27] Julian Martin Kunkel, Eugen Betke, Matt Bryson, Philip Carns, Rosemary Francis, Wolfgang Frings, Roland Laifer, and Sandra Mendez. Tools for Analyzing Parallel I/O. In Rio Yokota, Michèle Weiland, John Shalf, and Sadaf Alam, editors, *High Performance Computing*, Lecture Notes in Computer Science, pages 49–70, Cham, 2018. Springer International Publishing.

[28] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale deep learning for climate analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC ’18, pages 1–12, Dallas, Texas, November 2018. IEEE Press.

[29] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[30] Heshan Lin, Xiaosong Ma, Wuchun Feng, and Nagiza F. Samatova. Coordinating Computation and I/O in Massively Parallel Sequence Search. *IEEE Transactions on Parallel and Distributed Systems*, 22(4):529–543, April 2011.

[31] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA Tensor Core Programmability, Performance & Precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531, May 2018.

[32] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J. Pennycook, Kristyn Maschhoff, Jason Sewall, Nalini Kumar, Shirley Ho, Michael F. Ringenburg, Prabhat, and Victor Lee. CosmoFlow: Using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 1–11, Dallas, Texas, November 2018. IEEE Press.

[33] John McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, pages 19–25, December 1995.

[34] Jagan Meena, Simon Sze, Umesh Chand, and Tseung-Yuen Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9(1):526, 2014.

[35] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U-chupala, Yoshiki Tanaka, and Yuichi Kageyama. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv:1811.05233 [cs]*, March 2019.

[36] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8026–8037. Curran Associates, Inc., 2019.

[37] Sarunya Pumma, Min Si, Wu-Chun Feng, and Pavan Balaji. Scalable Deep Learning via I/O Analysis and Optimization. *ACM Transactions on Parallel Computing*, 6(2):1–34, July 2019.

[38] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, Cambridge, 2014.

[39] Ohad Shamir. Without-Replacement Sampling for Stochastic Gradient Methods. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 46–54. Curran Associates, Inc., 2016.

[40] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. UCF101: A Dataset of 101 Human Actions Classes From Videos in The Wild. *arXiv:1212.0402 [cs]*, December 2012.

[41] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective I/O in ROMIO. In *Proceedings. Frontiers '99. Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, Annapolis, MD, USA, 1999. IEEE.

[42] Shuai Wang, Bo Kang, Jinlu Ma, Xianjun Zeng, Mingming Xiao, Jia Guo, Mengjiao Cai, Jingyi Yang, Yaodong Li, Xiangfei Meng, and Bo Xu. A deep learning algorithm using CT images to screen for Corona Virus Disease (COVID-19). *medRxiv*, page 2020.02.14.20023028, April 2020.

[43] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. Yet Another Accelerated SGD: ResNet-50 Training on ImageNet in 74.7 seconds. *arXiv:1903.12650 [cs, stat]*, March 2019.

[44] Chih-Chieh Yang and Guojing Cong. Accelerating Data Loading in Deep Neural Network Training. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 235–245, December 2019.

[45] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image Classification at Supercomputer Scale. *arXiv:1811.06992 [cs, stat]*, December 2018.

[46] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 1–10, Eugene, OR, USA, August 2018. Association for Computing Machinery.

[47] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan 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 (MASCOTS)*, pages 145–156, Milwaukee, WI, September 2018. IEEE.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Clairvoyant Prefetching for Machine Learning I/O

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Böhringer	Roman

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Kilchberg, 23.07.2020	R. Böhringer

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.