Clairvoyant Prefetching for Distributed Machine Learning I/O

Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, Torsten Hoefler
Department of Computer Science, ETH Zürich, Switzerland

ABSTRACT

I/O is emerging as a major bottleneck for machine learning training, especially in distributed environments. Indeed, at large scale, I/O takes as much as 85% of training time. Addressing this I/O bottleneck necessitates careful optimization, as optimal data ingestion pipelines differ between systems, and require a delicate balance between access to local storage, external filesystems, and remote nodes. We introduce NoPFS, a machine learning I/O middleware, which provides a scalable, flexible, and easy-to-use solution to the I/O bottleneck. NoPFS uses *clairvoyance*: Given the seed generating the random access pattern for training with SGD, it can exactly predict when and where a sample will be accessed. We combine this with an analysis of access patterns and a performance model to provide distributed caching policies that adapt to different datasets and storage hierarchies. NoPFS reduces I/O times and improves end-to-end training by up to 5.4× on the ImageNet-1k, ImageNet-22k, and CosmoFlow datasets.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**.

KEYWORDS

Deep learning, high-performance computing, I/O

ACM Reference Format:
Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, Torsten Hoefler. 2021. Clairvoyant Prefetching for Distributed Machine Learning I/O. In *Supercomputing '21: The International Conference for High Performance Computing, Networking, Storage, and Analysis, November 14–19, 2021, St. Louis, MO.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Training deep neural networks (DNNs) is an increasingly important workload on supercomputers, as deep learning (DL) is adopted by more fields. Given the high cost of training, it is critical that every aspect be as efficient as possible [66, 74]. Extensive work has been done to optimize training [14], including dedicated hardware [44, 58], compilers [21, 29], optimizing operator primitives [22, 39], and communication infrastructure [10, 11, 27, 64, 73].

From the perspective of a DL framework, training a DNN involves three aspects: computation to execute the DNN; communication, to synchronize updates across nodes; and I/O, which provides the data and labels for training to each node. The vast majority of work on optimizing training has focused on computation and communication. Consequently, the performance bottleneck in training is shifting to I/O [63, 70]. Indeed, we find that when training ResNet-50 [34] on ImageNet [26] at scale, up to 85% of runtime is I/O overhead, and we observe similar trends in other datasets. As trends in compute capability continue with improving machine learning accelerators and datasets reach hundreds of millions [76] to billions [57] of samples and terabytes [3, 59, 67] to petabytes [2] in size, this I/O bottleneck will only be exacerbated.

Approach	System scalability	Dataset scalability	Full randomization	Hardware independence	Ease of use
Double-buffering (e.g., PyTorch [68])	✗	✓	✓	✗	✓
`tf.data` [1, 63]	✗	✓	✗	✗	✓
Data sharding (e.g., [50])	✓	✗	✗	✗	✓
DeepIO [79]	✓	✗	✗	✗	✓
LBANN data store [40, 67]	✓	✗	✓	✗	✗
Locality-aware loading [78]	✓	✓	✓	✗	✗
NoPFS (this paper)	✓	✓	✓	✓	✓

Table 1: Comparison of I/O frameworks.

It is challenging to optimize training I/O, as stochastic gradient descent (SGD) randomly accesses (typically small) data samples. This problem is especially acute for distributed training, where shared filesystem contention can be detrimental to performance. Existing frameworks often overlap I/O with computation to reduce its overhead, but this is no longer sufficient. Beyond this, ad hoc solutions such as limited lookahead and double-buffering [1, 23, 68], data sharding [30, 50], prestaging and in-memory caching [40, 67], or modified access patterns [78, 79] are used. These have significant limitations, including poor scalability, requiring extra hardware, neglecting parts of the storage hierarchy, or deviating from full-dataset randomization. All of these approaches can fail to fully utilize a machine's I/O subsystem (Tab. 1, Sec. 2).

To address the I/O bottleneck, we introduce a new I/O middleware framework, the Near-optimal PreFetching System, NoPFS. The key idea behind NoPFS is to use *clairvoyance* [13]: Given the seed for the pseudorandom number generator (PRNG) that generates an access stream, we know exactly which process will access a given sample when, arbitrarily far in the future. NoPFS analyzes the access stream to perform integrated prefetching and caching, rather than always reading from storage (Sec. 3). It combines this with a performance model-driven distributed caching policy that uses both on-node storage hierarchies (e.g., RAM, node-local SSDs) and distributed memory (Secs. 4, 5). This results in much-improved I/O performance, and overall improvements in runtime for ResNet-50 [34] on ImageNet [26] of up to 5.4×, up to 2.4× on the larger ImageNet-22k dataset, and 2.1× on CosmoFlow [59] (Sec. 7).

Using NoPFS requires no changes to deep learning frameworks and changes to only a few lines of code in existing training scripts, as it presents an iterator-style interface to accessing data like standard data loaders (Fig. 7). It can also automatically adapt to different

Correspondence to: ndryden@ethz.ch.

SC'21, November 14–19, 2021, St. Louis, MO

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

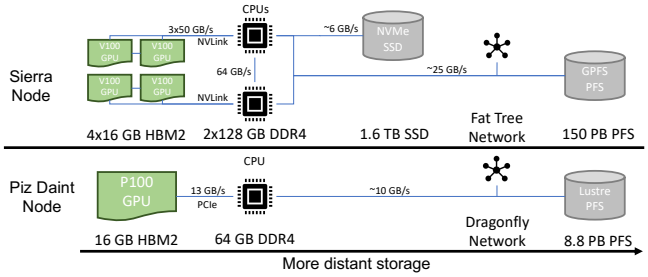


Figure 1: Storage hierarchies in modern supercomputers.

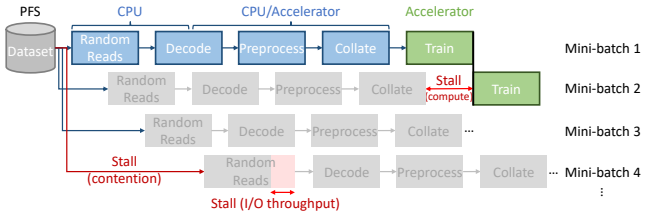


Figure 2: Overview of I/O pipelines and potential issues.

datasets and machines, being applicable both to small-scale research clusters and large supercomputers. Further, I/O subsystems are growing increasingly complex and differ between systems (Fig. 1), a trend set to continue with future systems such as DAOS [9] and Rabbit [62], making generic, performance model-driven systems even more attractive.

We additionally develop an I/O performance simulator to compare different I/O strategies in a variety of scenarios (Sec. 6). Beyond evaluating performance, this simulator can also be used to help design future systems for training workloads by analyzing which components (e.g., larger SSDs) have the largest impact on runtime.

When using NoPFS, I/O resources are fully utilized, alleviating the I/O bottleneck such that training is necessarily limited by the dataset and hardware. Our key contributions are:

- We identify clairvoyance as a key insight for optimizing I/O and use this to perform a probabilistic analysis of access patterns and produce a near-optimal mapping of data to cache hierarchies.
- We develop a performance model-driven distributed caching policy and implement it in NoPFS, an easy-to-use I/O middleware to optimize training I/O.
- We significantly reduce I/O overhead, improving overall training time on ImageNet, ImageNet-22k, and CosmoFlow by up to 5.4×.

2 BACKGROUND

Deep neural networks are almost always trained with variants of mini-batch SGD [17]. Training consists of many epochs; each epoch is a complete pass over the training dataset in a different, random order. The samples that make up a given mini-batch are randomly selected without replacement from the entire training dataset. This is typically implemented by assigning an index to each sample, randomly shuffling the indices each epoch, and then partitioning the shuffled indices into mini-batches. Hence, a given sample is accessed exactly once in each epoch. Given the seed used to shuffle the indices, we can exactly replicate the result of the shuffles, no matter the shuffle algorithm, and hence predict the access pattern, giving us *clairvoyance*. These access patterns hold across nearly all neural networks trained with mini-batch SGD. For distributed training, we assume a data-parallel regime [14], where a mini-batch is partitioned among workers.

2.1 Data Sharding

A relaxation of this regime sometimes used in practice is to assign each worker a shard (subset of the dataset) of data that fits in local storage, which may share samples with other workers (e.g., Kurth et al. [50]). This approach is typically adopted in order to mitigate the I/O overheads of shared storage by only using local storage. However, this has three major limitations: (1) The dataset must fit in the aggregate local storage. If it does not, then samples or even entire rare classes may be missed, impacting learning. Further, on many systems, local storage is small; e.g., Piz Daint [25] has 64 GB/node and Fugaku [71] only 32 GB/node, which must also hold the model and activations when training. (2) This can change the randomization performed by SGD, which may impact learning. It has been observed that full-dataset randomization and without-replacement sampling performs better [30, 32]. (3) This does not fully utilize the storage hierarchy, as it neglects distributed memory. For example, while accessing a node-local SSD may be faster than a contended PFS, it may be faster still to access samples from a remote node's memory, as modern network bandwidth (often 10+ GB/s) is higher than SSD read bandwidth (2–10 GB/s); an ideal solution would use both. In this work, we will focus on full-dataset randomization to avoid any issues with learning.

2.2 Machine Learning I/O Frameworks

I/O for training deep neural networks is a complex task and typically consists of multiple stages. At the highest level, "I/O" involves reading samples from storage, preprocessing them, where preprocessing may entail multiple stages itself, such as decoding images, tokenizing text, normalization, or data augmentation, and finally collating them into a mini-batch for training (see Fig. 2). Stalls at any point in this process can impact training time.

There are a number of solutions for optimizing the preprocessing stage, such as memory mapping optimizations [70]. In general, these are orthogonal to optimizations for the read stage. DALI [65] includes both preprocessing optimizations and a file cache. We focus primarily on optimizing this first stage, which we will refer to as "I/O". In this setting, we will assume that the dataset begins in a shared storage location such as a parallel filesystem (PFS), and that training workers run on compute nodes that all have access to the storage. This matches the MLPerf-HPC requirements [61].

We identify several key characteristics of I/O frameworks:

System scalability Whether additional resources can be productively used when scaling to many nodes.

Dataset scalability Whether arbitrarily large datasets (much larger than aggregate node storage) are supported.

Full randomization Whether sample selection is randomized over the entire dataset without replacement in each epoch.

Hardware independence Whether special hardware (e.g., node-local SSDs) is used if present, but is not required. Storage hierarchies are complex and often differ between systems (Fig. 1), making this especially important.

Ease of use Whether significant effort is needed to incorporate the framework in workflows.

We summarize existing approaches, along with NoPFS, according to these characteristics in Tab. 1. All of these approaches are capable of double-buffering, where fetching the next mini-batch is overlapped with computation, and using multiple threads to fetch and preprocess samples. This approach is taken by PyTorch's built-in `DataLoader` [68]. TensorFlow's `tf.data` [63] extends this with longer-range lookahead, but typically performs data shuffling only in a limited-size buffer instead of over the entire dataset. These two approaches have poor system scalability, as workers contend for access to shared storage. Data sharding is widely used in practice, generally with ad hoc data staging scripts, but is necessarily limited by available system storage. None of the existing approaches are hardware independent; either they require additional hardware, such as SSDs, or they neglect the hardware when it is available.

3 I/O ACCESS PATTERNS

We first review the prior work on prefetching and caching algorithms, and then analyze I/O access patterns in training. For caches, given the access stream R, the optimal schedule is given by Bélády's clairvoyant algorithm [13], which replaces the data that will not be needed for the longest time. However, it is much more challenging to derive an optimal schedule for integrated prefetching and caching [19]. There exist efficient algorithms for finding the optimal schedule in the case of a single processor and disk, and approximation algorithms for the case of a single processor with a parallel disk system [5–7, 20, 42, 48]. Unfortunately, finding an optimal schedule for the parallel disk case is NP-hard [8]. Similar work has been done in the context of caches for multi-core processors, where there are results for cache hierarchies, although optimal algorithms are again NP-hard [4, 15, 33, 45–47, 55]. Our case, where there are multiple processors each possibly with multiple levels of cache, is even more challenging.

Nevertheless, we can adapt ideas from these algorithms to our situation. It can be shown that any optimal prefetching and caching strategy for a single processor and disk must follow four rules [19]:

(1) **Optimal prefetching:** Every prefetch should fetch the next sample in R that is not in the cache.
(2) **Optimal replacement:** Every prefetch should discard the sample whose next use is furthest in the future.
(3) **Do no harm:** Never discard sample A to prefetch sample B when A will be used before B.
(4) **First opportunity:** Never prefetch-and-replace when the same operation could have been done previously.

Some of these rules can be generalized to the case of multiple disks [48], or relaxed while still producing good approximations [42]. NoPFS is able to implement Rule 1 exactly and approximates the remaining rules within a limited time horizon, using the fact that a sample is accessed exactly once per epoch.

Variable	Unit	Definition
R		Access sequence of a worker
N		Number of workers
E		Number of epochs
F		Number of samples in dataset
c	MB/s	Compute throughput
β	MB/s	Preprocessing rate
b_c	MB/s	Inter-worker network bandwidth
$t(\gamma)$	MB/s	Random agg. read throughput (with γ clients) of the PFS
p_j		Number of threads for prefetching to storage class j
d_j	MB	Capacity of storage class j
$r_j(p)$	MB/s	Random agg. read throughput of storage class j (p reader threads)
$w_j(p)$	MB/s	Random agg. write throughput of storage class j (p writer threads)
D	MB	Total local storage of a worker
s_k	MB	Size of sample k
S	MB	Size of dataset
B		Batch size
T		Number of iterations per epoch
$t_{i,f}$		Time elapsed when worker i consumes sample R_f

Table 2: Notation used throughout paper.

3.1 Distribution of Accesses

NoPFS utilizes a second key observation about the access pattern: Although each sample is read once per epoch, the number of times the same worker will read that sample over E epochs of training varies depending on the random seed. Exploiting this access frequency disparity allows us to devise better cache policies.

To formalize this, consider a fixed worker and sample, and let X_e be the probability that worker will access the sample in epoch e. For N workers and E epochs, we have that $X_e \sim \text{Bernoulli}(\frac{1}{N})$ and the access frequency X of this sample is $X = \sum_{e=1}^{E} X_e$. As the X_e are independent Bernoulli random variables with the same success probability, we have that $X \sim \text{Binomial}(E, \frac{1}{N})$. Thus the mean of the distribution is $\mu = \mathbb{E}[X] = \frac{E}{N}$ and the probability that the access frequency is greater than μ by a factor δ (and hence the sample will be accessed more often by the worker) is

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

However, we are primarily interested in the number of samples that will be accessed more often by a worker, which is the sum over all samples of $\mathbb{1}_{X>(1+\delta)\mu}$. Then, using that expectation is linear, the expected value is given by $F \cdot P(X > (1+\delta)\mu)$, where F is the size of the dataset. We verified that this works well using Monte Carlo simulations. As an example, consider a standard ImageNet-1k training run with $N = 16$, $E = 90$, $F = 1{,}281{,}167$, and $\delta = 0.8$. Our estimate gives an expected value of ~31,635: although each sample is read 6 times on average by a worker, around 31,635 samples will be accessed more than 10 times. The distribution from a Monte Carlo simulation is shown in Fig. 3. The number of samples accessed more than 10 times is 31,863, closely agreeing with the calculations.

As each sample is accessed exactly E times by fully-randomized SGD without replacement, if some worker access a sample more (or less) frequently, then some other worker must access it less (or more) frequently. We formalize this as follows:

Lemma 1. *If a worker accesses a sample $\lceil(1+\delta)\frac{E}{N}\rceil$ times (resp. $\lfloor(1-\delta)\frac{E}{N}\rfloor$ times), at least one other worker will access the sample at most $\lceil(\frac{N-1-\delta}{N-1})\frac{E}{N}\rceil$ (resp. at least $\lfloor(\frac{N-1+\delta}{N-1})\frac{E}{N}\rfloor$) times.*

Proof. Assume towards a contradiction that every other worker accesses the sample $\lceil(\frac{N-1-\delta}{N-1})\frac{E}{N}\rceil$+1 times for some $E, \delta \in [0, N-1]$,

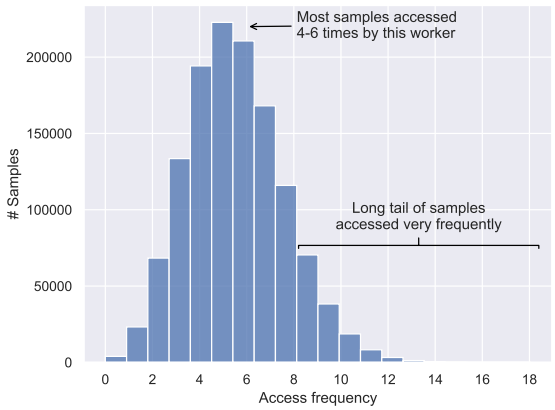


Figure 3: Simulation of access frequency for a single process (of 16) when training for 90 epochs on ImageNet-1k.

and $N > 1$. Then the total accesses to this sample are

$$\left\lceil (1+\delta) + \frac{E}{N} \right\rceil + (n-1)\left(\left\lceil \left(\frac{N-1-\delta}{N-1}\right)\frac{E}{N} \right\rceil + 1\right) \geq$$
$$(1+\delta)\frac{E}{N} + (N-1)\left(\left(\frac{N-1-\delta}{N-1}\right)\frac{E}{N} + 1\right) =$$
$$(1+\delta)\frac{E}{N} + E - (1+\delta)\frac{E}{N} + N - 1 =$$
$$E + N - 1 > E.$$

This contradicts that every sample is accessed exactly E times during training. The proof of the other bound is symmetric. □

4 PERFORMANCE MODELING

The I/O access frequency distribution allows us to identify frequently used samples to cache on a worker. We now turn to our performance model of training I/O, which will allow us to decide where to cache samples and where to fetch them from to minimize training time. These two analyses combined form the basis for NoPFS (Sec. 5). It also enables us to develop a simulator to compare I/O frameworks, identify bottlenecks, and help design future systems for training workloads (Sec. 6).

First we introduce some notation to define the compute environment; all associated quantities can be measured with simple benchmarks such as training microbenchmarks, STREAM [60], FIO [12], and IOR [37]. To simplify presentation, we will assume there is one worker per compute node (this is not necessary in practice).

Let there be N workers, where each worker i has:

- c [MB/s]: Compute throughput for training. This depends on the details of the neural network, hardware, and software. We model c as MB/s as it directly relates to I/O subsystem parameters; if it is known only in terms of samples/second, it can be approximated by multiplying this by the average file size. If samples are resized during preprocessing, the original size should be used.
- β [MB/s]: Data preprocessing rate.
- We will assume there is no network congestion.
- b_c [MB/s]: Inter-worker network bandwidth.
- $t(\gamma)$ [MB/s]: Random aggregate read throughput of the PFS, as a function of the number of readers γ. This depends on γ as PFS bandwidth is heavily dependent on the number of clients [24].

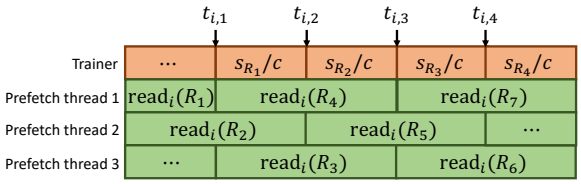


Figure 4: Performance model access times.

To account for the storage diversity present in current and upcoming systems, we will assume there are J distinct storage classes which group similar storage media. E.g., a storage class can represent RAM, SSDs, HDDs, shared global burst buffers, or emerging NVRAM technologies. Storage class 0 is defined to be the staging buffer, a (usually small) in-memory buffer that is shared with the machine learning framework. Storage class j is characterized by:

- d_j [MB]: Capacity of storage class j. The total local storage of a worker is therefore $D = \sum_{j=1}^{J} d_j$.
- $r_j(p)$ and $w_j(p)$ [MB/s]: Random aggregate read and write throughput for storage class j with p threads.
- p_j: Number of threads used for prefetching data to storage class j. We assume there is always at least one thread for prefetching to the staging buffer, i.e., $p_0 \geq 1$.

We model throughput as a function of p as for many storage devices, a single thread cannot saturate its bandwidth.

Let our training dataset consist of F samples, where sample k has size s_k. Each sample may have a different size. The size of the whole dataset is $S = \sum_{k=1}^{F} s_k$. We can have that $S > D$, where it is not possible for the dataset to be stored on a single worker, or $S > ND$, where the dataset cannot be stored across all workers. The mini-batch size is B and there are E epochs. One epoch consists of $T = \lfloor \frac{F}{B} \rfloor$ iterations (or $\lceil \frac{F}{B} \rceil$ if we keep the last, small iteration).

At iteration h, $1 \leq h \leq ET$, we process a batch $B^h \subseteq \{1, \ldots, F\}$ and worker i processes its local batch $B^{h,i} \subseteq B^h$. We write $b_i = |B^{h,i}|$. As each sample is read exactly once in an epoch, the sets B^k for $rT \leq k \leq (r+1)T$, for some $r \in \mathbb{N}$, are pairwise disjoint, which implies the same for the $B^{k,i}$. For data-parallelism, we have that $B^{h,1}, \ldots, B^{h,N}$ partition B^h. (Adapting this to other training regimes, e.g., model-parallelism, is straightforward.)

Lastly, we write $B_\ell^{h,i}$ to be the ℓth sample in worker i's hth batch. Then the worker's access stream is $R = (B_1^{1,i}, B_2^{1,i}, \ldots, B_{b_i}^{1,i}, B_1^{2,i}, \ldots)$.

We now define the key metric of our model, $t_{i,f}$, the time elapsed when worker i consumes R_f, the fth entry of R:

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

where $\text{avail}_i(f)$ is the time R_f is available in the staging buffer. This is illustrated in Fig. 4. Assuming threads are load balanced, we have

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

We define $\text{read}_i(k) = \text{fetch}_i(k) + \text{write}_i(k)$ as the time to read the kth sample into the staging buffer. Here, $\text{fetch}_i(k)$ is the time to fetch the sample into memory and $\text{write}_i(k)$ the time to preprocess and store it in the staging buffer. $\text{write}_i(k)$ does not depend on the

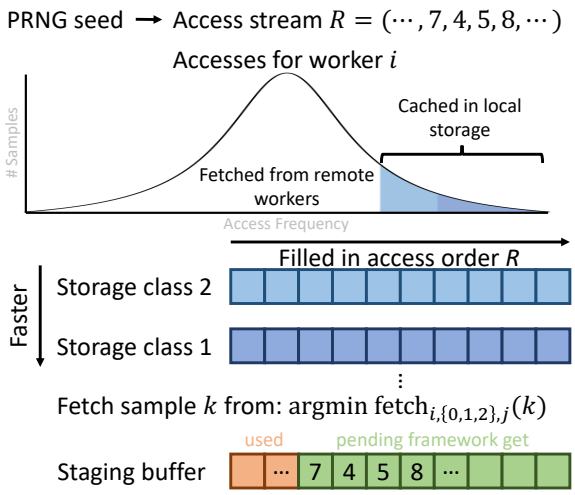


Figure 5: Overview of NoPFS's prefetching/caching policy.

data source, and is

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

where we assume preprocessing and writing can be pipelined in parallel. For fetching data, there are three cases, and we use the fastest applicable one:

(1) Reading from the PFS, while $\gamma - 1$ other workers do as well: $\text{fetch}_{i,0,0}(k) = s_k/(t(\gamma)/\gamma)$.
(2) Reading from another worker's storage class j: $\text{fetch}_{i,1,j}(k) = s_k/\min(b_c, r_j(p_j)/p_j)$.
(3) Reading from its local storage class j (assuming the sample is present): $\text{fetch}_{i,2,j}(k) = s_k/(r_j(p_j)/p_j)$.

This performance model drives runtime selection of data fetching and caching in NoPFS. Note, in practice, because of remote data fetching, we may not know the exact number of threads accessing a local storage class. However, this generally does not change the rank ordering due to disparities in speed of access.

5 NoPFS

We now present the design and implementation of NoPFS, which combines the aforementioned performance model with our analysis of access patterns to provide distributed caching and prefetching.

5.1 Design

NoPFS needs to answer several questions to implement its prefetching and caching policy: (1) Which samples should be fetched to the staging buffer when? (2) Where should these samples be fetched from? (3) Which samples should be assigned to which storage class, and what order should they be prefetched in? We will discuss each of these in turn; because NoPFS uses clairvoyance and a performance model, the solutions are near-optimal. The overall policy is summarized in Fig. 5.

As we know the PRNG seed, we can exactly compute R, and with this prefetch data in the correct access order into the staging buffer (satisfying Rule 1). Once a sample is read, a worker will access it again at the earliest in the next epoch, and every sample that follows in the current epoch is necessarily accessed earlier. Therefore, we can approximate Rules 2–4 by immediately dropping samples from the staging buffer after access, freeing up space for samples that (with high probability) will be accessed sooner.

While this determines which samples to fetch to the staging buffer at what time, we need to use our performance model to decide from where to fetch samples. Because we know R for each worker, every worker knows where every sample is cached, and we can select the location to fetch from that requires minimal time.

Finally, we define the strategy used by other storage classes. Suppose the worst case where a worker always waits before consuming a sample. Then the total training time is

$$t_{i,|R|} = \text{avail}_i(|R|) = \frac{\sum_{k=1}^{|R|}(\text{fetch}_i(R_k) + \text{write}_i(R_k))}{p_0}.$$

We fill the other storage classes to minimize this. If we ignore fixed terms in the strategy, we need to compute $\min \sum_{k=1}^{|R|} \text{fetch}_i(R_k)$. As we can select the fastest location to fetch from, this becomes

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

where j_r and j_ℓ are the fastest remote and local storage class of sample R_k, respectively. If a file is not available locally or at any remote worker, we define the respective throughput to be 0. Letting r_k be the access frequency of sample k and assuming a static system (i.e., samples are already loaded in storage classes and no parameters change), this becomes a sum over all samples:

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

Assuming that samples are similarly sized, we can conclude:

(1) When r_k is large (i.e., a worker accesses a sample frequently), we want $r_{j_\ell}(p_{j_\ell})$ to be large, and therefore should cache the sample in a fast local storage class.
(2) As $t(\gamma)/\gamma$ is often constant or decreasing with many readers, we want to minimize γ to reduce PFS contention. We also want $r_{j_r}(p_{j_r})$ to be large for samples where $r_{j_\ell}(p_{j_\ell})$ is small (i.e., samples not cached locally, or in slow storage). These imply samples should be well-distributed among workers.

Recall that the access frequency r_k varies for different k and Lemma 1 implies that when r_k is small on one worker, it will be large on at least one other worker (and vice versa). We thus use r_k to make the fetching decision: A worker fetches samples with the largest r_k to its fastest storage class, and so on for slower classes until either it has cached the entire dataset or filled its local storage.

The last step is to define the fetch order. Our analysis has thus far assumed all storage classes have already been filled, but this would require a potentially costly prestaging step that cannot be overlapped with training. We follow Rule 1 and use R to ensure we always fetch the samples in order of access.

5.2 Implementation

We now briefly describe the implementation of this design, summarized in Fig. 6. NoPFS consists of a core backend implemented in

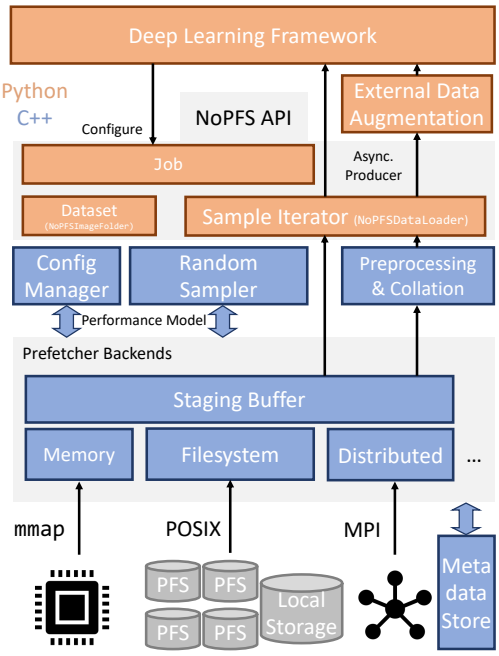


Figure 6: Overview of the NoPFS implementation.

C++ and a generic Python interface that exposes the functionality for integration with existing deep learning frameworks.

5.2.1 Python Interface. The Python interface provides the `Job` class, which represents the execution of a machine learning job on a particular dataset. A single Python process can run multiple jobs at the same time (e.g., training and validation). This only requires the user to specify a few parameters, such as the dataset, batch size, and the number of epochs. The random seed that generates the access sequence can either be specified manually by the caller or generated by the library.

Once initialized, the `Job` exposes two key features: `buffer_p`, a pointer to NoPFS's staging buffer, allowing zero-copy access to samples; and a `get` method, which returns samples and their labels, enabling iterator-style access to data.

It is easy to incorporate this into existing training pipelines. We provide convenience wrappers to replace the data loader and commonly used datasets in PyTorch. Using these, minimal changes are required to integrate NoPFS with existing PyTorch codebases, as we demonstrate in Fig. 7.

PyTorch data loading pipeline:

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

NoPFS data loading pipeline:

```
job = Job(data_dir, batch_size, num_epochs, 'uniform', drop_last)
dataset = NoPFSImageFolder(data_dir, job, data_transforms)
dataloader = NoPFSDataLoader(dataset)
```

Figure 7: PyTorch versus NoPFS data loading.

5.2.2 C++ Core. The core of NoPFS comprises a central manager, generic backends for storage and prefetching, and utilities. For simplicity, the parameters for our performance model are specified by a system-wide configuration file, with parameterized values (e.g., PFS bandwidth for a given number of readers) inferred using linear regression when the exact value is not available. This could be generalized to automatically determine performance parameters.

Storage backends need only implement a generic interface, and NoPFS currently supports filesystem- and memory-based storage backends, which are sufficient to support most storage classes (including RAM, SSDs, and HDDs). Additional backends (e.g., for key-value stores or databases) can easily be added.

For tracking samples, a metadata store keeps a catalog of locally cached samples. A distributed manager class handles all distributed operations among workers, using MPI for the underlying communication infrastructure. During setup, it is responsible for distributing a worker's access sequence R to all other workers (an `allgather`).

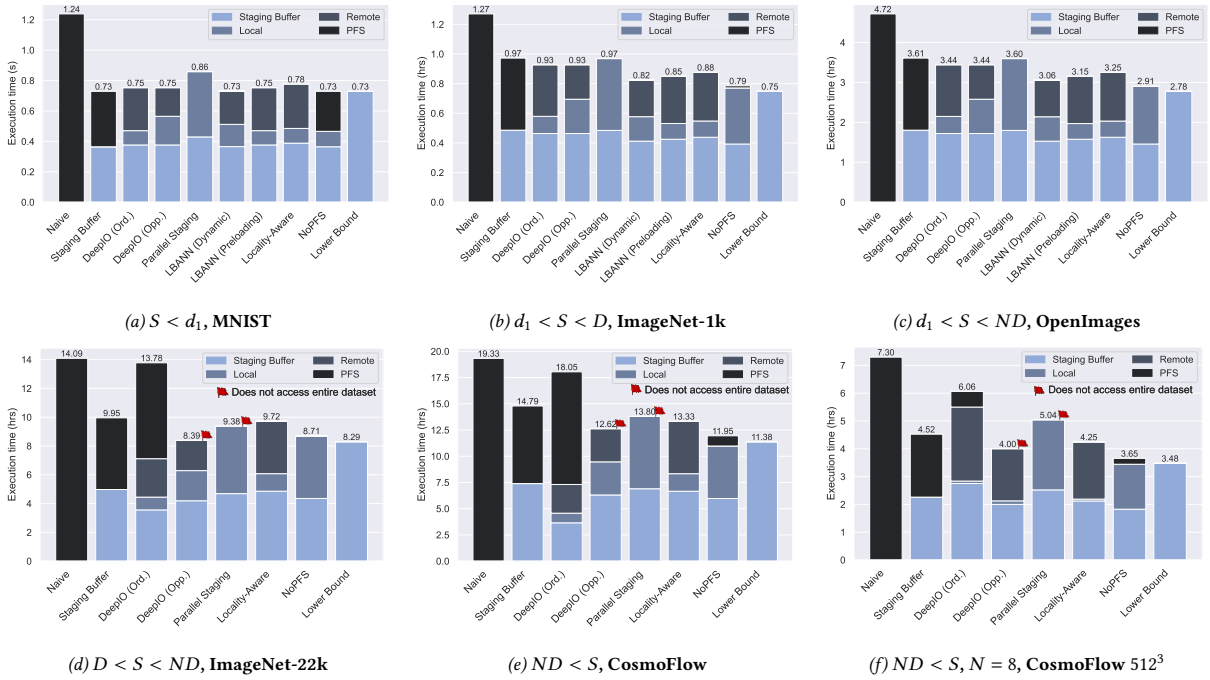
It also provides functionality for serving locally cached samples to and requesting samples from remote nodes. While it is always possible for a worker to know that a sample is not cached by any other worker, it is not possible (without additional metadata traffic) for a worker to know whether a worker that will cache a sample has successfully prefetched it. As requesting a remote sample that has not yet been cached results in wasted communication and increased stall time, we use a heuristic to estimate when a sample has been cached. Assuming samples are of comparable size and prefetching is load balanced, if the local prefetching has reached the corresponding access stream location, then the remote worker likely has, too. Note that the failure of this heuristic is not an error, as NoPFS detects such cases, but we wish to minimize such occurrences due to their performance impact. We confirmed that, in practice, there are very few false positives.

The core prefetching logic is managed by prefetcher backends, which implement all the logic for prefetching to a particular storage class. Adding a new prefetcher again only requires implementing a simple, generic interface. NoPFS provides a memory-based prefetcher and a filesystem-based prefetcher (which uses `mmap` to access files). We also implement a special prefetcher for the staging buffer, which is filled in a circular manner. This prefetcher coordinates with the Python interface via a producer/consumer queue to ensure that the consumer knows when samples are available, and that the prefetcher knows when samples have been consumed (and therefore can be replaced). If a prefetcher for a local storage class finds that a sample that should be present has not yet been fetched, that prefetcher will retrieve and cache the sample itself, helping to smooth out load imbalance.

Finally, the configuration, storage classes, and prefetchers are managed by a prefetcher manager class, which coordinates the different components. We also provide convenience utilities, including an optimized, OpenCV-based [18] image preprocessing pipeline, and batch collation directly into a pinned memory buffer, which we observed could be a bottleneck otherwise.

6 PERFORMANCE SIMULATOR

We developed a performance simulator based on our performance model to evaluate different data loading strategies. The simulator



(a) $S < d_1$, **MNIST** *(b)* $d_1 < S < D$, **ImageNet-1k** *(c)* $d_1 < S < ND$, **OpenImages**

(d) $D < S < ND$, **ImageNet-22k** *(e)* $ND < S$, **CosmoFlow** *(f)* $ND < S$, $N = 8$, **CosmoFlow** 512^3

Figure 8: Performance simulation results. Stacked bars show the proportion of time for each I/O location.

supports arbitrary dataset, system, and I/O strategy configurations. We do not aim for a precise simulation of training, but rather to capture the relative performance of different I/O strategies. To this end, we adapt the performance model from Sec. 4, where compute/communication throughput is based on c, and assume I/O is overlapped to the greatest extent possible. We focus on four cases:

(1) $S < d_1$: The dataset fits into the first storage class (typically RAM) of each worker. This should not be a challenging situation, but is nevertheless important, as it occurs with small datasets or workers with large amounts of RAM.
(2) $d_1 < S < D$: The dataset fits in the aggregate storage of a worker. This scenario is interesting, as while a worker can cache the entire dataset, it must use multiple storage classes to do so.
(3) $D < S < ND$: The dataset can be cached in the aggregate storage of all workers. This requires workers to exploit distributed caching and to minimize the number of PFS accesses.
(4) $ND < S$: The dataset is too large to be cached even by the aggregate storage of all workers. While this is an uncommon scenario today when using many workers, it is interesting to examine, especially as dataset sizes grow in the future. Further, this scenario already occurs when large datasets are used on small training clusters.

We study the following policies:

(1) `Perfect`: This simulates the case where no stalls occur and provides a lower bound, although it is not realistic in practice.
(2) `Naive`: Loading from the PFS with no prefetching or caching.
(3) `StagingBuffer`: This fills a staging buffer according to the reference string, fetching data from a given location and dropping it after it is consumed. When configured to prefetch data from the PFS, this simulates the double buffering or `tf.data` policies.
(4) `DeepIO`: This simulates the ordered and optimistic modes for DeepIO [79]. The latter mode may change the access order.
(5) `ParallelStaging`: This simulates data sharding, which also changes the access order, as only locally-available samples are accessed by a worker.
(6) `LBANN`: This simulates the LBANN data store [40] (dynamic and preloading approaches). As this only caches data in memory, it will fail if the dataset exceeds the aggregate worker memory.
(7) `LocalityAware`: This simulates the locality-aware approach of Yang and Cong [78]. When using this policy, we reorder batches at the beginning of the simulation to correspond to the logic described in their paper.
(8) `NoPFS`: NoPFS's policy (Sec. 5).

6.1 Simulation Results

For brevity, we report simulation results for a single setup simulating a small cluster in Fig. 8. Each plot summarizes the execution time, and the stacked bars give the proportion of execution time spent fetching from a particular storage class. We use $N = 4$ workers, each on a dedicated node with a compute throughput of $c = 64$ MB/s, a preprocessing rate $\beta = 200$ MB/s, and an inter-worker bandwidth $b_c = 24{,}000$ MB/s. We configured a 5 GB staging buffer, and two further storage levels representing 120 GB of RAM and a 900 GB local SSD. We use eight, four, and two prefetcher threads per storage level, and set $r_0(8) = 111$ GB/s, $r_1(4) = 85$ GB/s, and $r_2(2) = 4$ GB/s. For PFS read throughput, we set $t(1) = 330$ MB/s, $t(2) = 730$ MB/s, $t(4) = 1{,}540$ MB/s, and $t(8) = 2{,}870$ MB/s. These choices were based on benchmarks of the Lassen supercomputer [54].

We simulate a set of representative datasets; datasets with different filesizes are assumed to be distributed normally and we vary the μ and σ parameters and the number of samples, F, to match. A per-worker batch size of $B = 32$ was used, except for the large CosmoFlow datasets, where $B = 16$ and $B = 1$, respectively.

Scenario 1 ($S < d_1$, $\mu = 0.76$ KB, $\sigma = 0$, $F = 50{,}000$, 40 MB, MNIST [52]): This is representative of small research datasets commonly used in practice. There is relatively little difference in performance for most policies, and they are close to the lower bound. The exception is `Naive`, which is 1.7× slower than the best-performing policy, illustrating the importance of proper I/O handling.

Scenario 2 ($d_1 < S < D$, $\mu = 0.1077$ MB, $\sigma = 0.1$, $F = 1{,}281{,}167$, 135 GB, ImageNet-1k [72]; and $\mu = 0.2937$ MB, $\sigma = 0.2$, $F = 1{,}743{,}042$, 500 GB, OpenImages [51]): Here, NoPFS is the best-performing policy, and is very close to the theoretical lower bound. There are several key factors behind this performance: NoPFS does not require an initialization phase (in contrast to data staging), reduces PFS reads (whereas `StagingBuffer` policies always read from the PFS), and utilizes all available resources (in contrast to the LBANN data store, which uses only RAM).

Scenario 3 ($ND < S$, $\mu = 0.1077$ MB, $\sigma = 0.2$, $F = 14{,}197{,}122$, 1,500 GB, ImageNet-22k [26]): In this scenario, the dataset exceeds the aggregate storage capacity of each worker. Further, the LBANN data store no longer supports the dataset, as it is larger than aggregate RAM. The DeepIO ordered mode performs poorly here, since it fetches uncached samples from the PFS and does not consider access frequency for assigning samples. NoPFS again performs well and approaches the lower bound. DeepIO and parallel data staging are also able to perform well, as they never access the PFS, mitigating the impact of the large dataset size. However, *they no longer access the entire dataset*, significantly impacting potential accuracy during training.

Scenario 4 ($ND < S$, $\mu = 17$ MB, $\sigma = 0$, $F = 262{,}144$, 4 TB, CosmoFlow [59]; and $N = 8$, $\mu = 1{,}000$ MB, $\sigma = 0$, $F = 10{,}000$, 10 TB, CosmoFlow 512^3 [67]): We simulate two versions of the CosmoFlow dataset, which are representative of emerging scientific workloads. The standard CosmoFlow dataset, part of MLPerf-HPC [61], consists of a large number of 128^3 samples. That dataset is derived from the CosmoFlow 512^3 dataset, which consists of a smaller number of large, 512^3 samples that are sliced to produce the 128^3 samples. We use 8 workers for CosmoFlow 512^3. In both cases, the dataset size exceeds the storage capacity of the cluster. Performance is similar to Scenario 3, but at larger scale, indicating NoPFS is able to strong scale well with dataset size and cluster resources while still providing access to the full dataset.

6.2 Environment Evaluation

In addition to comparing I/O policy performance, our simulator can also be used to quantify the impact of changes to a system on training time. This can be used to identify promising hardware upgrades or when designing new systems to meet training requirements.

To illustrate this, we consider the ImageNet-22k dataset from Scenario 3 with the NoPFS policy and vary the system configuration, assuming 5× compute and preprocessing throughput, which is representative of future machine learning accelerators. The lower bound on runtime is 1.06 hours. We first simulate using only a staging buffer of size 1, 2, 4, or 5 GB, which all resulted in runtimes of 1.64 hours, indicating that the staging buffer is not the limiting factor in this configuration; we fix it at 5 GB. We next considered configurations with 32, 64, 128, 256, or 512 GB of RAM and 128, 256, 512, or 1024 GB of SSD as additional storage classes. The performance for these configurations is illustrated in Fig. 9.

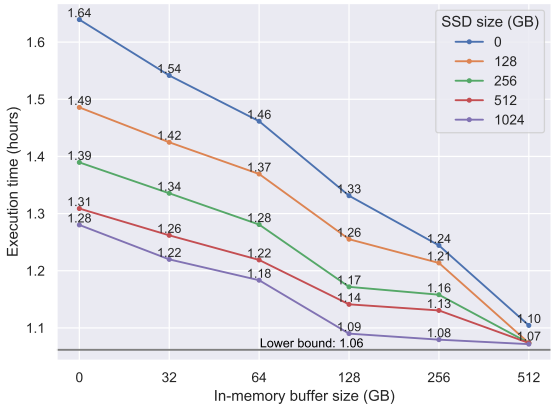


Figure 9: Performance simulation for ImageNet-22k.

We observe that, while the best performance is achieved by maximizing total storage, different combinations of storage can often be used to achieve a given performance level if other factors (e.g., cost) need to be optimized for. Notably, if memory is maximized, then SSD size becomes less relevant. Alternatively, if memory is expensive, it can be compensated for with additional SSD storage. This demonstrates why it is critical that an I/O framework be able to automatically adapt to many different storage backends.

7 EVALUATION

We now experimentally validate NoPFS and compare it to PyTorch using both its built-in `DataLoader` and DALI [65]. Our experiments use the Piz Daint [25] and Lassen [54] supercomputers. Fig. 1 provides system details (Lassen uses the same architecture as Sierra). All runs begin with data at rest on a PFS, in line with MLPerf-HPC guidelines [61]. We perform each run in a separate job allocation to mitigate caching effects. On Lassen, we use one rank per GPU.

Frameworks We use PyTorch 1.7 with NCCL2 for all PyTorch benchmarks. For each model, we endeavored to provide a highly-optimized baseline and all runs use the same training implementation. We evaluated four different frameworks for I/O:

- `PyTorch`: The built-in PyTorch `DataLoader` and `DistributedSampler`, with multiple prefetching and preprocessing threads.
- `DALI`: DALI 0.31.0 for prefetching and preprocessing. DALI only supports x86 CPUs, so we only report results for Piz Daint.
- `NoPFS`: Our NoPFS implementation, integrated into PyTorch.

On Lassen, a NoPFS rank (four per node) uses a 5 GiB staging buffer with eight prefetching threads, 25 GiB of RAM with four prefetching threads, and 300 GiB of SSD with two prefetching threads. On Piz Daint, it uses a 5 GiB staging buffer with four prefetching threads and 40 GiB of RAM with two prefetching threads. (We used as much memory as possible without OOM errors.) To ensure preprocessing is not a bottleneck, we extended PyTorch and NoPFS

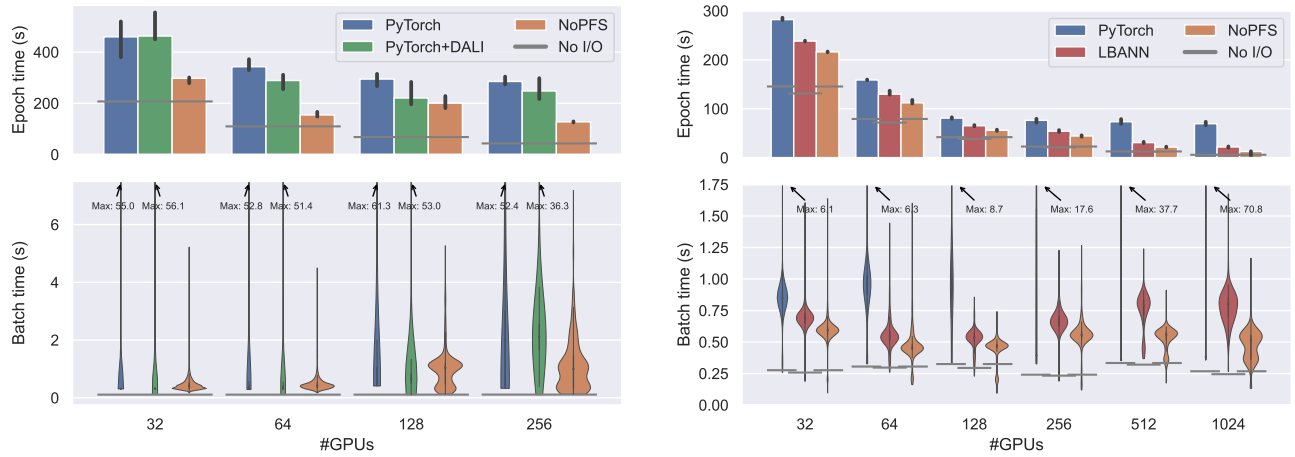


Figure 10: Epoch & batch time for training ResNet-50 on ImageNet-1k on Piz Daint (left) and Lassen (right) (excl. epoch 0). NoPFS is up to 2.2× faster than PyTorch on Piz Daint and up to 5.4× faster on Lassen; it is also up to 1.7× faster than LBANN.

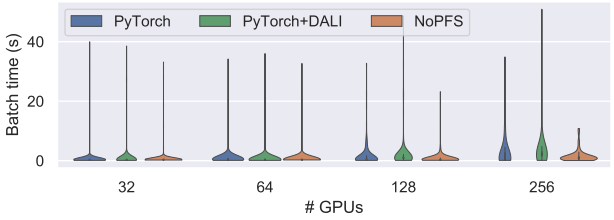


Figure 11: Epoch 0 batch time for ImageNet-1k on Piz Daint.

to perform some data augmentation and conversion on a separate CUDA stream on GPU; DALI does this automatically.

- `LBANN`: LBANN, using its data store in dynamic mode. In this mode, each sample is cached in memory by the worker that reads it first. LBANN requires sufficient memory for its cache.

Datasets We use three datasets, of significantly different sizes and representative of different tasks:

- ImageNet-1k [72]: We train ResNet-50 on ImageNet-1k as a standard baseline. ImageNet-1k consists of 1,281,167 images and 1,000 classes, totalling 135 GB. We use the standard data layout, with one directory per class containing all images of that class. We use a per-GPU batch size of 64 on Piz Daint and 120 on Lassen. Standard data augmentation (random resizes, crops, flips, and normalization) is performed.
- ImageNet-22k [26]: We train ResNet-50 on the larger ImageNet-22k dataset, which consists of 14,197,103 samples and 21,841 classes, totalling 1.3 TB. This is more representative of larger, emerging datasets used for unsupervised or semisupervised pre-training. The configuration is otherwise identical to ImageNet-1k.
- CosmoFlow [59]: We use the MLPerf-HPC [61] CosmoFlow model and dataset. The data consists of 262,144 simulated 3D universes of size $128 \times 128 \times 128$ and four channels, stored in 16-bit integer format, totalling 4 TB. Instead of the original HDF5 data format, we converted the data to a simple binary format. As HDF5 requires locking to serialize I/O accesses, we found it did not perform well, with median batch times of 3.2 s. We use a per-GPU batch size of 16 and log normalization on Lassen.

Synthetic data lower bound To provide a lower bound for training with no I/O and minimal perturbation, we use synthetic data. We pregenerate random samples in RAM of the appropriate size and data type and use them for training. The decoding, preprocessing, augmentation, and other aspects of training are otherwise identical to regular training. We report this as "No I/O" in plots. As this measurement is experimental, some results are slightly faster. Since LBANN has slightly different performance than PyTorch, its "No I/O" performance is measured separately.

7.1 I/O Performance

We first compare the training performance of each framework. We evaluate ImageNet-1k on both Lassen and Piz Daint (Fig. 10), and the remaining datasets on Lassen (Figs. 14 and 15). We report median time per epoch with a 95% confidence interval, using 10 epochs for ImageNet-1k and CosmoFlow and 3 epochs for ImageNet-22k. We also show violin plots of the per-batch time, skipping the first epoch (which has consistently high variance due to initial data access). NoPFS consistently has the fastest runtimes and small variance, even on Piz Daint, where the variance of other frameworks is high.

ImageNet-1k On Piz Daint, NoPFS is 2.2× faster than the PyTorch `DataLoader` and 1.9× faster than PyTorch+DALI on 256 GPUs. On Lassen, it is 5.4× faster than PyTorch and 1.7× faster than LBANN on 1024 GPUs. We observe consistent scaling, except for 128 nodes on Piz Daint, where significant system noise in the NoPFS run resulted in worse performance relative to NoPFS at 64 GPUs. Despite this, NoPFS is still faster than others. On Piz Daint, DALI offers a relatively small performance improvement over the default PyTorch `DataLoader`, likely because its data augmentation pipeline is already well-optimized, including offloading some augmentation to GPU. Due to PFS contention limiting I/O, PyTorch does not scale beyond 256 GPUs on Lassen. Despite LBANN being a faster framework than PyTorch in this benchmark (as its no I/O baseline indicates), NoPFS in PyTorch is still able to outperform it. At small

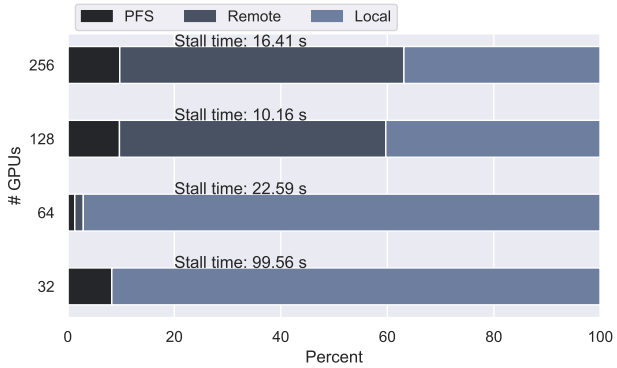


Figure 12: NoPFS cache stats for ImageNet-1k on Piz Daint.

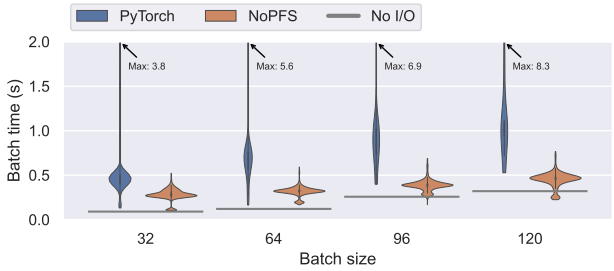


Figure 13: Varying batch sizes for training ResNet-50 on ImageNet-1k on 128 GPUs on Lassen (excl. epoch 0).

scale, the difference in performance is minimal, but it becomes more significant at larger scale. This is because LBANN's data store uses a simple first-touch policy for caching samples, and caches each sample in only one location. Hence, at larger scales, many samples need to be fetched from remote nodes. While this avoids issues of PFS contention, it is suboptimal compared to NoPFS's access frequency-based caching.

In per-batch runtimes, NoPFS exhibits significantly less variance at all scales than other methods. Its batches are also fast much more consistently. This demonstrates a key performance advantage of NoPFS: reducing tail events where read performance is catastrophically slow due to system contention by using local or remote caches. After the first epoch, PyTorch and DALI exhibit tail events an order of magnitude larger than NoPFS. We also examined the batch times in the first epoch (Fig. 11) on Piz Daint. NoPFS shows comparable or only slightly lower variance to the other methods, as all must initially access data from the PFS, although NoPFS mitigates this with its prefetching. However, for PyTorch and DALI, the variance here is comparable to the variance in subsequent epochs: without caching, it is always "the first epoch" for a data loader.

To break down the source of NoPFS's improvements, Fig. 12 presents the stall time and the percent of staging buffer prefetches that were from local storage, a remote node's cache, or the PFS, aggregated over all epochs. Stall time decreases at larger scales, as NoPFS is able to strong scale to take advantage of additional storage across the cluster. The fetch locations also demonstrate how NoPFS adapts to changing cluster conditions. At smaller scales, the PFS is under less contention, and NoPFS is able to prefetch into on-node memory quickly. Further, as the number of GPUs increases, each node sees a smaller portion of the dataset, making the prefetching task easier. However, beyond 64 GPUs, it becomes slower to read from the PFS, and NoPFS instead fetches samples from remote nodes that have already cached them.

Impact of Batch Size It is common to vary the batch size when training, depending on how one wishes to trade off memory and learning versus GPU utilization. To study this effect, we compare NoPFS with PyTorch when training ImageNet-1k on 128 GPUs on Lassen (Fig. 13). We observe that NoPFS is faster at every batch size (the runtime per batch necessarily increases with larger batch sizes, due to more samples being processed). Further, while the variance in runtime stays roughly constant for NoPFS, for PyTorch it increases significantly with larger batches, due to additional I/O pressure caused by each rank fetching more data.

ImageNet-22k & CosmoFlow Both of these datasets demonstrate similar performance trends as for ImageNet-1k on Lassen. At 1024 GPUs, NoPFS is 2.4× faster on ImageNet-22k and 2.1× faster on CosmoFlow. This demonstrates how NoPFS is able to automatically adapt to very different datasets: Either many more samples (ImageNet-22k) or much more data (CosmoFlow). For CosmoFlow in particular, NoPFS is very close to the no I/O lower bound. NoPFS also automatically takes advantage of SSDs to cache parts of the CosmoFlow dataset at small scale, when the aggregate node memory is insufficient to hold the dataset.

The batch times for CosmoFlow also exhibit an interesting bimodal distribution. This is because the samples are all the same, large size (16 MB), leading to significantly different runtimes depending on where the sample is fetched from.

Discussion While NoPFS shows large performance improvements across systems, scales, and datasets, there is still a gap between its performance and the no I/O lower bound. We profiled the ImageNet-1k training with 32 GPUs on Lassen and observed that NCCL allreduces took up to 2× longer when performing I/O than without I/O. We believe this is due to increased contention when performing I/O, as I/O threads interfere with NCCL's communication threads and I/O traffic goes over the same network as allreduces. This problem is more acute for ImageNet than for CosmoFlow, as the former uses much larger batches and smaller, variable-sized samples, resulting in much more frequent I/O requests. Indeed, this "I/O noise" presents a more general problem: since training is bulk synchronous due to the allreduces in each mini-batch, I/O noise becomes a barrier to scalability [35, 69]. NoPFS helps to significantly reduce this, but better characterizing and mitigating I/O noise (e.g., dedicated I/O cores or storage networks) are important future work.

In general, NoPFS's distributed caching means that samples are read from the PFS as few times as necessary; typically only once for an entire training run when the dataset fits in the aggregate storage. This has two advantages: NoPFS suffers from less noise due to contention on the PFS, and it has a lower impact on other jobs that may be using the PFS in a shared cluster environment. Further, as NoPFS scales, it can take advantage of additional distributed memory. Perhaps counterintuitively, because of very high-speed networks and better random-access performance, reading from *remote* memory can be faster than reading from a local SSD. While remote accesses

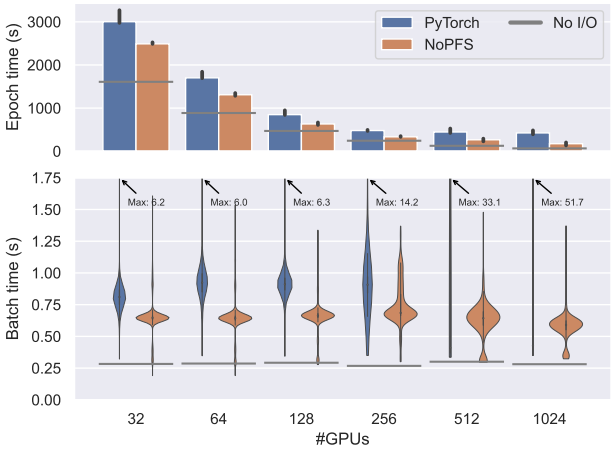


Figure 14: Epoch & batch time for ImageNet-22k on Lassen. NoPFS is up to 2.4× faster.

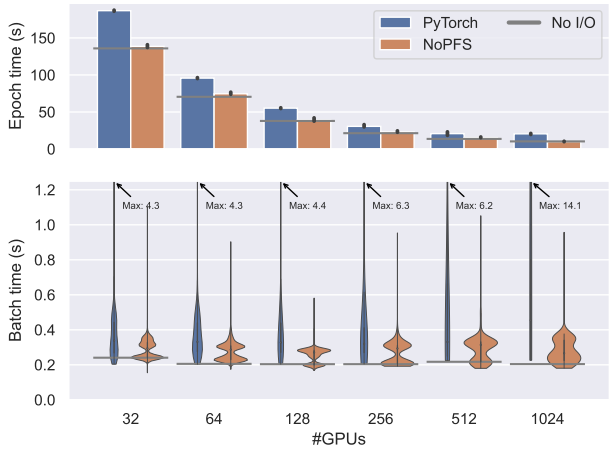


Figure 15: Epoch & batch time for CosmoFlow on Lassen. NoPFS is up to 2.1× faster.

increase network traffic, which can interfere with allreduces, not using NoPFS *would still require similar network communication*, since the PFS is accessed over the same network in our systems, while NoPFS avoids PFS contention. Overall, NoPFS introduces very little overhead (compared to a standard I/O framework, it only needs to compute the access sequence in advance, which is fast) and in practice demonstrates large performance improvements of 2×–5×.

7.2 End-to-end Training

Finally, we performed end-to-end training of ImageNet-1k using 256 GPUs on Lassen. We use a batch size of 32 samples per GPU, for a global batch size of 8192, and follow the learning procedure in Goyal et al. [30]. The top-1 validation accuracy over time for both NoPFS and PyTorch are shown in Fig. 16. In line with our benchmarks, we achieve a 1.42× speedup over the standard PyTorch `DataLoader` while achieving state-of-the-art accuracy. Both runs exhibit similar learning curves, albeit with slight variation due to different random seeds for network parameters. (Note, due to the speedup, NoPFS's curve is compressed.)

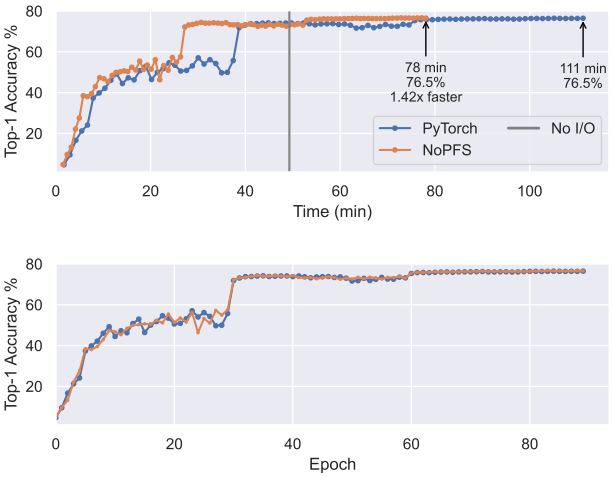


Figure 16: Training ResNet-50 on ImageNet-1k using 256 GPUs on Lassen, accuracy vs time (top) and epochs (bottom).

8 RELATED WORK

Beyond work on optimizing ML training I/O (see Sec. 2.2), there has been work on optimizing specific aspects or infrastructure. Pumma et al. [70] optimizes LMDB, Caffe's [41] I/O subsystem, to address issues in `mmap` I/O request scheduling. Chowdhury et al. [24] study the performance of the BeeGFS filesystem for deep learning workloads. Chien et al. [23] examine the impact of multi-threading in TensorFlow's I/O pipeline. Data preprocessing and augmentation can also be a bottleneck during training, as it is typically executed by CPUs, which may be unable to keep up with accelerators. Optimized pipelines such as DALI [65] attempt to address this with careful engineering and by splitting preprocessing between CPU and GPU.

Beyond these, efficient distributed I/O has long been studied in the context of scientific computing applications [36, 53, 77]. High-performance networks and RDMA have also been used to disaggregate memory and improve I/O performance. Infiniswap [31] used RDMA for remote memory paging. Key/value stores can leverage RDMA [43] or OpenSHMEM [28] for improved performance. A similar set of work exists for distributed filesystems, which also leverage non-volatile memory, including the Hadoop Distributed Filesystem [38], Octopus [56], and Crail [75]. Additionally, distributed and hierarchical caching has been studied in other contexts, such as for video-on-demand content [49] and content delivery networks [16].

9 CONCLUSIONS

Clairvoyance has long been an idea used in theoretical studies of prefetching and caching, but has been difficult to translate to practical applications with complex I/O access patterns. With machine learning, where the access pattern is random, but predictable given the random seed that generates it, there is now an application that fully benefits from this. Using clairvoyance, we make a probabilistic analysis of access patterns and show that there is almost always an

imbalance in the frequency a worker accesses a particular sample, which we combine with a performance model to drive a hierarchical caching and prefetching policy. NoPFS provides a simple, powerful interface that can be used in existing training pipelines to improve their I/O performance and reduce overall runtime.

As the compute throughput of accelerators continues to grow faster than that of data movement, the cost of I/O—and the importance of optimizing it—will only increase. Further, storage hierarchies are only getting deeper and more complicated, necessitating dedicated infrastructure to fully utilize them. Our work here serves as an initial step toward incorporating more detailed analyses of I/O into runtime frameworks. Future directions could include NUMA- and topology-awareness for data fetching; dynamic cache management, where samples can migrate between caches; and better characterizing I/O noise. We expect that by building on clairvoyance and other insights, the I/O bottleneck can be addressed.

ACKNOWLEDGMENTS

The authors thank Marc Snir for discussions inspiring this line of research, and Tim Moon, Quincey Koziol, John Ravi, and Suren Byna for feedback. This project received funding from the European Research Council (ERC) under the European Union's Horizon 2020 program (grant agreement MAELSTROM, No. 955513). N.D. is supported by the ETH Postdoctoral Fellowship. T.B.N. is supported by the Swiss National Science Foundation (Ambizione Project #185778). Experiments were performed at Livermore Computing and the Swiss National Supercomputing Center.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

[2] Franklin Abodo, Robert Rittmuller, Brian Sumner, and Andrew Berthaume. 2018. Detecting Work Zones in SHRP 2 NDS Videos Using Deep Learning Based Computer Vision. In *17th IEEE International Conference on Machine Learning and Applications (ICMLA)*.

[3] Sami Abu-El-Haija, Nisarg Kothari, Joonseok Lee, Paul Natsev, George Toderici, Balakrishnan Varadarajan, and Sudheendra Vijayanarasimhan. 2016. YouTube-8M: A large-scale video classification benchmark. *arXiv preprint arXiv:1609.08675* (2016).

[4] Kunal Agrawal, Michael A Bender, Rathish Das, William Kuszmaul, Enoch Peserico, and Michele Scquizzato. 2021. Tight Bounds for Parallel Paging and Green Paging. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. 3022–3041.

[5] Susanne Albers and Markus Büttner. 2005. Integrated prefetching and caching in single and parallel disk systems. *Information and Computation* 198, 1 (2005).

[6] Susanne Albers, Naveen Garg, and Stefano Leonardi. 2000. Minimizing stall time in single and parallel disk systems. *Journal of the ACM (JACM)* 47, 6 (2000).

[7] Susanne Albers and Carsten Witt. 2001. Minimizing stall time in single and parallel disk systems using multicommodity network flows. In *Approximation, Randomization, and Combinatorial Optimization: Algorithms and Techniques*.

[8] Christoph Ambühl and Birgitta Weber. 2003. Parallel Prefetching and Caching is NP-hard. (2003).

[9] Argonne Leadership Compute Facility. 2021. Aurora. https://www.alcf.anl.gov/aurora

[10] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 2017. S-Caffe: Co-designing MPI runtimes and Caffe for scalable deep learning on modern GPU clusters. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 193–205.

[11] Ammar Ahmad Awan, Karthik Vadambacheri Manian, Ching-Hsiang Chu, Hari Subramoni, and Dhabaleswar K Panda. 2019. Optimized large-message broadcast for deep learning workloads: MPI, MPI + NCCL, or NCCL2? *Parallel Comput.* 85 (2019), 141–152.

[12] Jens Axboe. 2021. fio - Felxible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html

[13] László A. Bélády. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal* 5, 2 (1966).

[14] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4 (2019).

[15] Michael A Bender, Roozbeh Ebrahimi, Jeremy T Fineman, Golnaz Ghasemiesfeh, Rob Johnson, and Samuel McCauley. 2014. Cache-adaptive algorithms. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. 958–971.

[16] Sem Borst, Varun Gupta, and Anwar Walid. 2010. Distributed caching algorithms for content distribution networks. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*.

[17] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *Siam Review* 60, 2 (2018).

[18] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

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

[20] Pei Cao, Edward W Felten, and Kai Li. 1994. Application-Controlled File Caching Policies.. In *USENIX Summer*. 171–182.

[21] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An end-to-end optimization stack for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[22] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[23] Steven WD Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. 2018. Characterizing deep-learning I/O workloads in TensorFlow. In *International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*.

[24] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. 2019. I/O characterization and performance evaluation of BeeGFS for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[25] CSCS. 2020. Piz Daint. https://www.cscs.ch/computers/piz-daint/

[26] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[27] Nikoli Dryden, Naoya Maruyama, Tim Moon, Tom Benson, Andy Yoo, Marc Snir, and Brian Van Essen. 2018. Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems. *Workshop on Machine Learning in HPC Environments (MLHPC)* (2018).

[28] Huansong Fu, Manjunath Gorentla Venkata, Ahana Roy Choudhury, Neena Imam, and Weikuan Yu. 2017. High-performance key-value store on OpenSHMEM. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 559–568.

[29] Google. 2020. XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla

[30] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch SGD: Training ImageNet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[31] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[32] Mert Gürbüzbalaban, Asu Ozdaglar, and Pablo A Parrilo. 2019. Why random reshuffling beats stochastic gradient descent. *Mathematical Programming* (2019), 1–36.

[33] Avinatan Hassidim. 2010. Cache Replacement Policies for Multicore Processors. In *Innovations in Computer Science*. 501–509.

[34] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. Characterizing the influence of system noise on large-scale applications by simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. 1–11.

[36] Mark Howison, Quincey Koziol, David Knaak, John Mainzer, and John Shalf. 2010. Tuning HDF5 for Lustre file systems. In *Workshop on Interfaces and Abstractions for Scientific Data Storage*.

[37] IOR team. 2021. HPC IO benchmark repository. https://github.com/hpc/ior

[38] Nusrat Sharmin Islam, Md Wasi-ur Rahman, Xiaoyi Lu, and Dhabaleswar K Panda. 2016. High performance design for HDFS with byte-addressability of NVM and RDMA. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–14.

[39] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. Data Movement Is All You Need: A Case Study on Optimizing Transformers. In *Proceedings of the Fourth Conference on Machine Learning and Systems (MLSys)*.

[40] 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, et al. 2019. Parallelizing training of deep generative models on massive scientific datasets. In *IEEE International Conference on Cluster Computing (CLUSTER)*.

[41] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[42] Wei Jin, Rakesh D Barve, and Kishor S Trivedi. 2002. A simple characterization of provably efficient prefetching algorithms. In *Proceedings of the International Conference on Dependable Systems and Networks*.

[43] Jithin Jose, Hari Subramoni, Miao Luo, Minjia Zhang, Jian Huang, Md Wasi-ur Rahman, Nusrat S Islam, Xiangyong Ouyang, Hao Wang, Sayantan Sur, et al. 2011. Memcached design on high performance RDMA capable interconnects. In *2011 International Conference on Parallel Processing*. 743–752.

[44] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*.

[45] Shahin Kamali and Helen Xu. 2020. Multicore Paging Algorithms Cannot Be Competitive. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 547–549.

[46] Shahin Kamali and Helen Xu. 2021. Beyond worst-case analysis of multicore caching strategies. In *Symposium on Algorithmic Principles of Computer Systems (APOCS)*. 1–15.

[47] Anil Kumar Katti and Vijaya Ramachandran. 2012. Competitive cache replacement strategies for shared cache environments. In *International Parallel and Distributed Processing Symposium (IPDPS)*. 215–226.

[48] Tracy Kimbrel and Anna R Karlin. 2000. Near-optimal parallel prefetching and caching. *SIAM Journal on computing* 29, 4 (2000).

[49] Christian Koch, Johannes Pfannmüller, Amr Rizk, David Hausheer, and Ralf Steinmetz. 2018. Category-aware hierarchical caching for video-on-demand content on YouTube. In *Proceedings of the 9th ACM Multimedia Systems Conference (MMSys)*.

[50] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, et al. 2018. Exascale deep learning for climate analytics. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[51] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, et al. 2018. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982* (2018).

[52] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998).

[53] Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Robert Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. 2003. Parallel netCDF: A high-performance scientific I/O interface. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[54] Livermore Computing. 2020. Lassen. https://hpc.llnl.gov/hardware/platforms/lassen

[55] Alejandro López-Ortiz and Alejandro Salinger. 2012. Paging for multi-core shared caches. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*. 113–127.

[56] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 773–785.

[57] Dhruv Mahajan, Ross Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. 2018. Exploring the limits of weakly supervised pretraining. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[58] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. NVIDIA tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*.

[59] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook, et al. 2018. CosmoFlow: Using deep learning to learn the universe at scale. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

[60] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2, 19–25 (1995).

[61] MLCommons. 2020. MLPerf HPC Training Rules. https://github.com/mlperf-hpc/training_policies/blob/hpc-0.5.0/hpc_training_rules.adoc

[62] Timothy Prickett Morgan. 2021. Livermore Converges a Slew of New Ideas for Exascale Storage. https://www.nextplatform.com/2021/03/09/livermore-converges-a-slew-of-new-ideas-for-exascale-storage/

[63] Derek G Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. 2021. tf.data: A Machine Learning Data Processing Framework. *arXiv preprint arXiv:2101.12127* (2021).

[64] Nvidia. 2020. NVIDIA Collective Communications Library. https://developer.nvidia.com/nccl

[65] Nvidia. 2020. NVIDIA Data Loading Library. https://developer.nvidia.com/DALI

[66] OpenAI. 2018. AI and Compute. https://openai.com/blog/ai-and-compute/

[67] Yosuke Oyama, Naoya Maruyama, Nikoli Dryden, Erin McCarthy, Peter Harrington, Jan Balewski, Satoshi Matsuoka, Peter Nugent, and Brian Van Essen. 2020. The case for strong scaling in deep learning: Training large 3D CNNs with hybrid parallelism. *IEEE Transactions on Parallel and Distributed Systems* (2020).

[68] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[69] Fabrizio Petrini, Darren J Kerbyson, and Scott Pakin. 2003. The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing (SC)*. 55–55.

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

[71] RIKEN Center for Computational Science. 2021. About Fugaku. https://www.r-ccs.riken.jp/en/fugaku/about/

[72] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015).

[73] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799* (2018).

[74] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics (ACL)*.

[75] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. 2017. Crail: A High-Performance I/O Architecture for Distributed Data Processing. *IEEE Data Eng. Bull.* 40, 1 (2017), 38–49.

[76] Chen Sun, Abhinav Shrivastava, Saurabh Singh, and Abhinav Gupta. 2017. Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*.

[77] Rajeev Thakur, William Gropp, and Ewing Lusk. 1999. Data sieving and collective I/O in ROMIO. In *Seventh Symposium on the Frontiers of Massively Parallel Computation*.

[78] Chih-Chieh Yang and Guojing Cong. 2019. Accelerating Data Loading in Deep Neural Network Training. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*.

[79] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. 2018. Entropy-aware I/O pipelining for large-scale deep learning on HPC systems. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.