Lobster: Load Balance-Aware I/O for Distributed DNN Training

Jie Liu, Bogdan Nicolae, Dong Li

Lobster: Load Balance-Aware I/O for Distributed DNN Training

Jie Liu
jliu279@ucmerced.edu
University of California, Merced
Merced, USA

Bogdan Nicolae
bnicolae@anl.gov
Argonne National Laboratory
Chicago, USA

Dong Li
dli35@ucmerced.edu
University of California, Merced
Mercced, USA

ABSTRACT

The resource-hungry and time-consuming process of training Deep Neural Networks (DNNs) can be accelerated by optimizing and/or scaling computations on accelerators such as GPUs. However, the loading and pre-processing of training samples then often emerges as a new bottleneck. This data loading process engages a complex pipeline that extends from the sampling of training data on external storage to delivery of those data to GPUs, and that comprises not only expensive I/O operations but also decoding, shuffling, batching, augmentation, and other operations. We propose in this paper a new holistic approach to data loading that addresses three challenges not sufficiently addressed by other methods: I/O load imbalances among the GPUs on a node; rigid resource allocations to data loading and data preprocessing steps, which lead to idle resources and bottlenecks; and limited efficiency of caching strategies based on pre-fetching due to eviction of training samples needed soon at the expense of those needed later. We first present a study of key bottlenecks observed as training samples flow through the data loading and preprocessing pipeline. Then, we describe Lobster, a data loading runtime that uses performance modeling and advanced heuristics to combine flexible thread management with optimized eviction for distributed caching in order to mitigate I/O overheads and load imbalances. Experiments with a range of models and datasets show that the Lobster approach reduces both I/O overheads and end-to-end training times by up to 1.5× compared with state-of-the-art approaches.

ACM Reference Format:
Jie Liu, Bogdan Nicolae, and Dong Li. 2022. Lobster: Load Balance-Aware I/O for Distributed DNN Training. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545090

1 INTRODUCTION

Deep Neural Networks (DNNs) are rapidly gaining traction in both industry and scientific computing, driven by the accumulation of massive data. In science, for example, instruments that collect data at GB/s and 100+ TB/day present a wide range of learning opportunities. We thus see significant interest in deploying DNNs on high-performance computing (HPC) systems in order to enable rapid learning in domains such as computational fluid dynamics [6], power grids [7], and molecular dynamics [40]. Various approaches [10, 28] for training DL models on massive data have been developed: coarse-grain parallelization on multiple nodes using data-parallel, model-parallel, pipeline-parallel, and hybrid techniques; fine-grain parallelization on many-core architectures by constructing and scheduling execution graphs at the tensor level; and low-level optimizations of operators [14] and communication primitives [4]. Most such work is targeted at alleviating the computational overhead needed to run iterations that perform forward and backward passes on mini-batches of training data, as well as the communication costs associated with synchronizing subtasks across devices and nodes.

However, before a mini-batch can be processed, it needs to be assembled through a complex pipeline that involves data loading, caching and pre-processing (decoding, augmentation, batching). This pipeline overlaps with the training itself, aiming to improve resource utilization and hide the overhead of assembling the mini-batches. However, despite this overlap, data loading and preprocessing often become a bottleneck [8, 19, 26, 30, 39], with reports of overheads of up to 72% of end-to-end training time [26, 30]. With ever-increasing accumulation of training data, data loading is likely to become yet more costly, prompting the need for scalable solutions to mitigate these overheads.

Unsurprisingly, several efforts have emerged to reduce the I/O overheads of data loading, such as double-buffering [1, 15, 31], data sharing [20, 29], domain specific caching [8, 19, 24, 39], and model relocation instead of data shuffling [27, 29]. However, despite significant progress, several challenges remain.

First, there can be a high degree of load imbalance across GPUs. For example, in a data-parallel training iteration, in which each GPU works on a different mini-batch, differences in the location of training samples (i.e., local cache vs. remote storage) lead to differences in data fetch costs, which in turn lead to some GPUs loading their data more slowly than others. As all GPUs must cooperate to average their gradients during the backward pass, these stragglers ultimately slow all GPUs. Thus, approaches that are agnostic to fine-grain data load imbalances across individual GPUs are insufficient.

Second, data loading competes for resources with the other stages of the pipeline. Focusing on the data loading alone and lacking coordination with the other stages of the pipeline may cause a resource utilization imbalance that further amplifies the I/O load imbalance. For example, if a fixed number of threads is allocated to each stage of the pipeline, then a bottleneck in one stage will lead to idle threads in the other stages that instead could have used to alleviate the bottleneck. Thus, it is important to coordinate thread management in order to avoid resource wastage.

Third, the caching of training samples in the node-local memory hierarchy can suffer from inefficient eviction. Caching, essential for reducing the I/O overheads associated with accessing a storage

repository, is often implemented in a distributed fashion: each compute node exposes its local cache to other compute nodes, greatly reducing the need for the compute nodes as a group to interact with the repository. By using a pseudo-random number generator to sample the training data, it is possible to obtain foreknowledge of the order in which the training samples will be accessed in future iterations. State-of-the-art approaches leverage such foreknowledge to prefetch training samples, further reducing I/O overheads. However, prefetching inevitably causes cache evictions, which may lead to suboptimal behavior if samples that will be accessed in the more distant future replace samples needed sooner.

To address the above challenges, we propose Lobster, a holistic data loading I/O runtime for distributed DNN training. Lobster distinguishes between the I/O load of each individual GPU at fine granularity and coordinates the I/O operations of the GPUs at the node level, flexibly allocating available I/O bandwidth and threads as needed to reduce I/O load imbalance.

Lobster also coordinates the data loading and preprocessing stages of the pipeline, flexibly allocating threads between them to reduce bottlenecks. This coordination is achieved through the use of performance modeling, which we combine with reuse distance theory to design efficient eviction policies for distributed caching of the training samples. Such an optimized eviction policy complements state-of-the-art distributed caching approaches based on prefetching by avoiding the undesirable effect of evicting training samples that are needed in the near future in order to make room for prefetched samples that are needed later. We show that this method increases the cache hit ratio by 14.3% compared with state-of-the-art prefetching approaches such as that used in NoPFS [8].

Thanks to these contributions, Lobster is able to maximize GPU and cache utilization, thus hiding the overheads of data loading and enabling high performance and scalable end-to-end DNN training.

- We characterize the performance (especially I/O performance) across 64 GPUs in a production environment for distributed DNN training, highlighting the I/O load imbalance across GPUs and frequent performance bottleneck shifts between data loading/preprocessing pipeline and the training process. This study reveals new opportunities for I/O performance optimization that are not considered by state-of-art approaches (Section 3).
- We propose a thread management strategy to coordinate the resource usage between data loading and preprocessing in the training pipeline (Section 4.1), as well as to mitigate the I/O load imbalance between GPUs (Section 4.2).
- We introduce a holistic performance model that bridges the thread management strategy with a distributed caching proposal that features prefetching support and optimized eviction based on reuse distance (Section 4.3).
- We design and implement a heuristic strategy to solve the optimization problem resulting from the performance model. This strategy consists of two phases (prefetching and eviction) and guides both the allocation of the threads and the distributed caching (Section 4.4).
- We evaluate Lobster on a 64-GPU (NVIDIA A100) cluster and compare its performance with the state-of-the-art PyTorch I/O [31], DALI [44], and NoPFS [8] systems on several DNN models and training datasets. Our experiments show end-to-end training speed-ups of 1.3×–2.0× and cache hit ratio improvements of 14.3%–38.7% (Section 5).

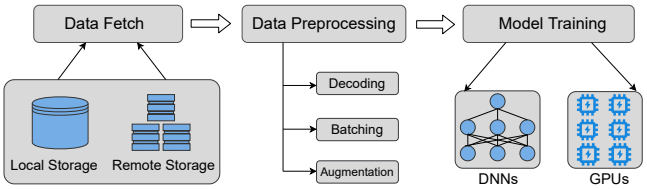

Figure 1: DNN training pipeline.

2 BACKGROUND

DNN training is an iterative process: first, the answer to an input is obtained in a forward pass over all layers. Then, in a backward pass, the difference (gradients) between the predicted and actual result ("ground truth") is used to update the weights layer by layer in reverse order. This process repeats for a large number of iterations until the DNN model has converged. Typically, multiple passes over the whole training data are required. Thus, iterations are grouped into epochs, each of which represents a full pass.

The input of each iteration is a *mini-batch*, which is obtained by random sampling of the training data. For efficiency reasons, in practice a pseudo-random number generator is used to shuffle the training samples, after which they are accessed in the shuffled order and grouped together as mini-batches. Since the seed of the pseudo-random number generator is known in advance, the I/O access pattern necessary to read the training samples can be made fully deterministic [8].

Before executing the forward and backward pass, the DNN training pipeline includes a data loading and preprocessing state, as illustrated in Figure 1. Data loading is responsible for prefetching and caching the training samples (which is possible thanks to the I/O access pattern being deterministic), while data preprocessing is responsible for additional transformations: decoding, augmentation, batching. All these stages in the pipeline are overlapping, which optimizes the resource utilization. The data preprocessing can be performed on either the CPU or the GPUs. For the purpose of this work, we assume the preprocessing is performed on the CPU, while the training is performed on the GPUs. This is a common scenario [2, 19, 21, 25, 26, 42] that makes efficient use of heterogeneous compute resources.

In order to scale the DNN training, multiple nodes equipped with multiple GPUs are used, as illustrated in Figure 2. The most common approach to achieve this is *data parallelism*, i.e., training the same DNN model replica on multiple GPUs with different mini-batches, then averaging the gradients during the backward pass. In this case, the GPUs co-located on the same node share a node-local cache. If a training sample is not available in the node-local cache, it can be retrieved either from the external storage repository (typically a parallel file system or PFS) or, if available, from the cache of a different node. The latter requires the implementation of a distributed cache but improves I/O latency significantly for several reasons: (1) the bandwidth between compute nodes is higher than the I/O bandwidth between a single compute node and the

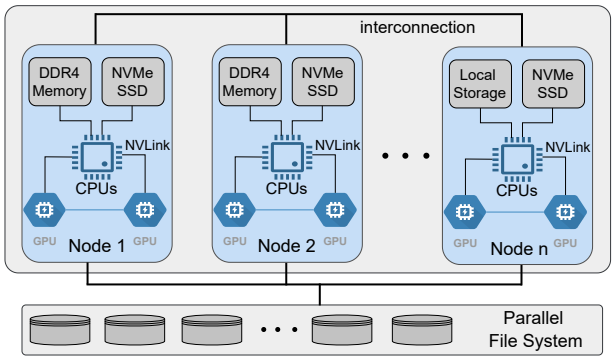

Figure 2: The storage hierarchy for distributed training in our environment.

PFS; (2) the aggregated I/O bandwidth of the PFS is limited and becomes a bottleneck when multiple compute nodes compete for it; (3) the PFS is not optimized for I/O access patterns that involve small randomly scattered reads necessary to retrieve the training samples.

In this paper, we assume a data-parallel training that makes use of a distributed cache. However, it is important to note that our proposal works in general for other DNN training scenarios as well (e.g., different DNN models sharing the same training data, alternatives to distributed caching like for example KV-stores, or even single-node DNN training). Our goal is to optimize node-local caches such that they can serve multiple co-located GPUs efficiently, when considering the challenges discussed in Section 1: (1) data load imbalance across the GPUs; (2) lack of coordination between the stages of the pipeline; (3) sub-optimal cache eviction due to deterministic prefetching.

3 MOTIVATION

We now motivate our approach by examining the challenges introduced in Section 1 in detail. To this end, we run a series of experiments that profile the performance of the DNN training pipeline and discuss our observations.

A detailed description of the experimental setup is available in Section 5.1. Using this experimental setup, we train a ResNet50 model on the ImageNet-1K dataset using PyTorch 1.8 as the DNN runtime and DALI [44], an industry-standard state of art data loading library. We use a data-parallel setup deployed on eight nodes, for a total of 64 GPUs. Since the stages of the training pipeline are overlapping and we are interested in studying the bottlenecks, we measure the duration of the delays caused by each stage along the critical path.

Figure 3 shows detailed results for three GPUs: two co-located on the same node, and the third on a different node. We omit the first epoch (because the caches need to warm up and therefore the behavior is different compared to the rest of the epochs) and focus on 24 iterations (out of 562) of the second epoch: eight each in the beginning, middle, and end—enough to capture a recurring pattern throughout the epoch. We make the following observations:

Observation 1: There is data load imbalance across GPUs. GPUs are often idle during an iteration, but not because they are waiting for their own data loading and pre-processing stages, which are faster than, and therefore fully overlap with, training. The problem is rather that other GPUs have longer data loading and preprocessing stages, which causes them to become stragglers and delay the start of the training stage. As each GPU needs to perform the same amount of work during the training stage, the stragglers cause other GPUs to sit idle while they wait to average the gradients during the backward pass. For example, during iteration 7, GPU0 of Node1 and GPU1 of Node2 are less loaded than GPU1 of Node1. Their GPU idle time takes 73% and 12% of the total iteration time, respectively. Compared with iteration 2, where there is no data load imbalance, iteration 7 is 3× slower. Our results show that such data load imbalances occur frequently: in 65.3% of our 562 iterations.

Observation 2: Data loading overheads vary frequently and irregularly across iterations, leading to the performance bottleneck shifting among the stages of the pipeline. Since the pipeline overlaps the stages, the slowest stage becomes a bottleneck. Ideally, data loading and preprocessing should never become a bottleneck. However, not only does this happens frequently with state-of-art approaches, but it is difficult to predict: during the same iteration, on some GPUs the training stage is the performance bottleneck, while the opposite is true on the other GPUs. Similarly, on the same GPU, the bottleneck can shift between data loading and training across iterations and exhibit an irregular pattern. This can happen on any GPU. In this experiment, we did not observe the preprocessing stage becoming a bottleneck by itself, however, this can happen and was reported by other studies [26]. When data loading is the bottleneck, training performance suffers significant slowdown. For example, in the case of GPU0 on Node0, in the two iterations where data loading is the bottleneck, its duration is 3× longer than the training stage—an observation that is explained by the fact that remote I/O to the external repository and/or the local caches of other compute nodes is orders of magnitude slower that local I/O. Furthermore, this effect is also correlated with load imbalance: whenever the performance bottleneck shifts in a GPU, other GPUs tend to exhibit load imbalance. Thus, when data loading is the bottleneck, it tends to exhibit a bursty pattern.

Observation 3: The preprocessing stage does not benefit from an arbitrarily large number of threads. Data preprocessing is characterized by a streaming memory access pattern: training samples are continuously arriving in batches, and each training sample can be further split into sub-domains (e.g. regions in an image). The computations are typically embarrassingly parallel, therefore they can considered a bag of tasks to be assigned to threads. Varying the number of threads changes data parallelism and memory bandwidth consumption, which in turn impacts performance. To study this effect, we vary the number of preprocessing threads and measure the preprocessing throughput (decoding/decompression and data augmentation). As can be observed in Figure 6, the preprocessing throughput peaks at 6 threads, after which it flattens and even slightly becomes worse. This effect has been observed by others as well [5, 22]. For data preprocessing, intensive memory bandwidth consumption is the major performance bottleneck when the number of threads is large. Furthermore, excessive memory bandwidth consumption also impacts the other stages in the pipeline. Therefore, it must be avoided.

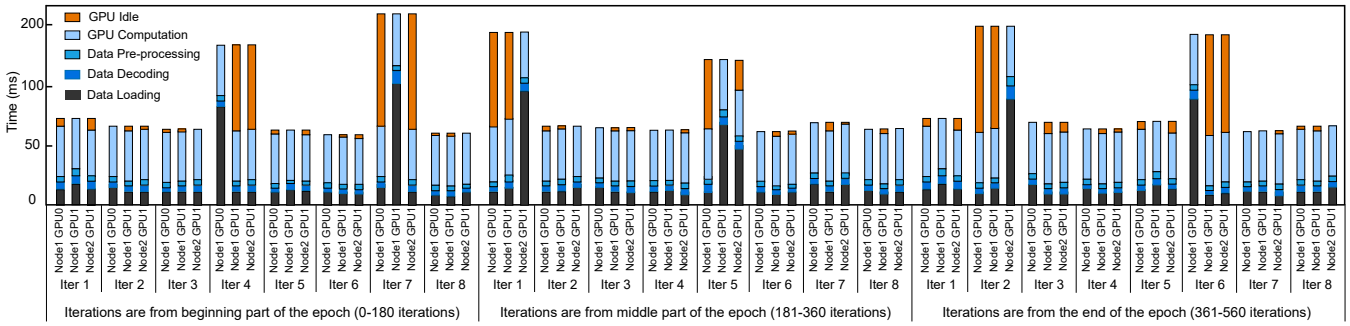

Figure 3: Execution time breakdown for the training pipeline on three GPUs, two on one node and the third on a second node.

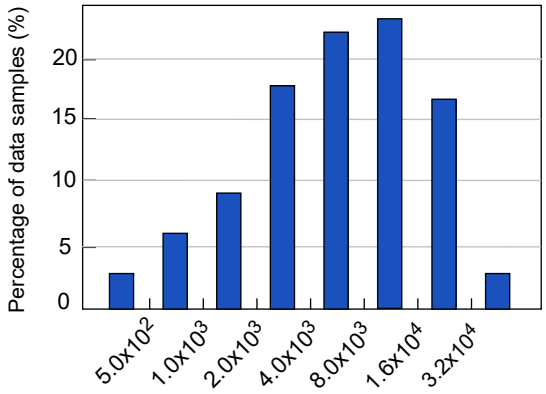

Figure 4: Histogram of the reuse distance of the training samples, measured in terms of numbers of iterations (X-axis)

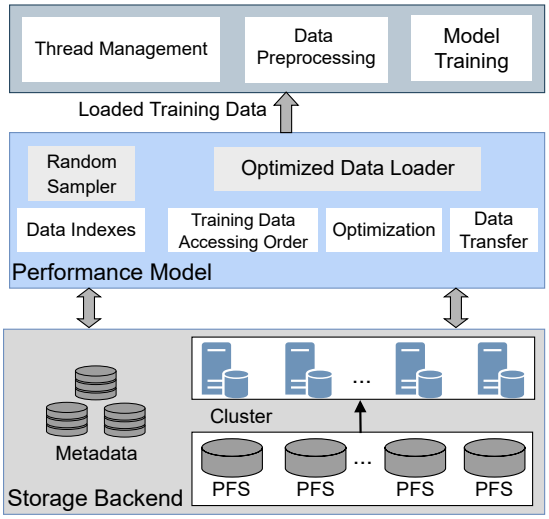

Figure 5: Overview of Lobster.

Observation 4: Many training samples have a long reuse distance. During data-parallel training, each GPU processes a different mini-batch. This means that each training sample cached on a compute node at iteration i may be reused the first time either by the same or a different GPU co-located on the same node at iteration j. We call $j - i$ the *reuse distance*. Studying the reuse distance of the training samples is important to understand how well the cache of each compute node is utilized. Figure 4 shows the histogram of reuse distance of data samples accessed by GPUs for Node1. We observe that many training samples have a long reuse distance. Here, when two memory accesses to a sample are separated by at least one epoch away, we call it "long". For example, 80% of the training samples have the reuse distance larger than 1,000 iterations.

Implications. Observation 1 indicates that the data load imbalance is frequent and leads to stragglers. Thus we need a fine-grained load balancing strategy that is aware of individual GPUs. Observation 2 is correlated to Observation 1 and points to a frequent shift of the performance bottleneck among the stages of the pipeline due to I/O bursts. Therefore, it is important to (1) optimize the utilization of the node-local cache to avoid remote I/O; (2) coordinate with the other stages of the pipeline and allocate more I/O threads to data loading when remote I/O cannot be avoided. Observation 3 indicates that the preprocessing stage does not benefit from an arbitrarily large number of threads and it can be even detrimental to allocate too many threads to it due to high memory bandwidth consumption. Therefore, it is important to determine the minimum number of threads needed to reach the peak preprocessing throughput and not exceed it. Observation 4 indicates that we can leverage long reuse distance of training samples to optimize the utilization of the node-local cache. In particular, we can explore new eviction policies that coordinate with prefetching to avoid the undesirable situation in which the training samples that are needed in the near future are evicted at the expense of the prefetched training samples that are needed later.

4 DESIGN

Based on the observations summarized in Section 3, we propose Lobster, a holistic data loading I/O runtime for distributed DNN training. Lobster uses the following high-level strategy to balance the work of the different stages of the training pipeline between different GPUs: (1) decide the number of data preprocessing threads (Section 4.1); (2) given the number of data preprocessing threads, when the data loading is not a performance bottleneck of the training pipeline, decide the number of data loading threads per iteration

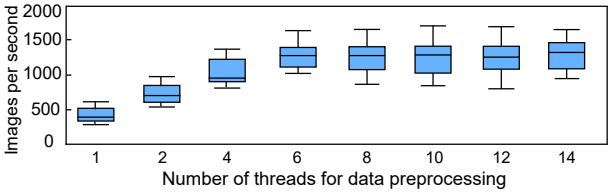

Figure 6: The impact of number of preprocessing threads (X-axis) on data preprocessing throughput (Y-axis)

for each GPU (Section 4.2); (3) given the number of data preprocessing threads, when the data loading is a performance bottleneck of the training pipeline, use performance modeling and run an heuristic algorithm to decide the number of data loading threads to balance I/O between GPUs in the same node (Sections 4.3 and 4.4). The heuristic algorithm considers the reuse distance of the training samples, which is used to coordinate with the prefetching and improve the overall hit ratio of the node-local cache. This strategy is implemented as illustrated in Figure 5.

Lobster addresses (1) by throttling thread-level parallelism in preprocessing so that it can redirect threads for data loading. To decide the number of preprocessing threads, Lobster predicts the preprocessing throughput based on a piece-wise linear regression model. For (2), Lobster introduces a multi-queue data structure to distinguish I/O between GPUs, and assigns threads to GPUs in proportion to data loading intensity. For (3), Lobster formulates the problem of deciding data loading threads as an optimization problem, and uses a computation-efficient heuristic algorithm to solve it. We now examine each of these aspects in detail.

4.1 Flexible Preprocessing Thread Management

We decide the number of data preprocessing threads based on two goals: (1) the combined duration of preprocessing and data loading should be smaller than the training stage; (2) need to redirect threads to improve the performance of the data loading stage.

Given a batch of training samples, we use the following two-step algorithm to meet the above goals. **Step 1:** predict the preprocessing throughput using the optimal number of preprocessing threads (that reaches the peak preprocessing throughput, as explained in Section 3). The prediction is based on performance modeling, as detailed below. Then, we use the methods in Sections 4.2 and 4.3 to decide the number of threads allocated for the data loading stage. This aims to reach goal (1). **Step 2:** as long as goal (1) is not reached and the preprocessing stage is not a performance bottleneck, take away one thread from the preprocessing stage and make it available for data loading. The frequency of running this algorithm can be adjusted to reach a trade-off where we avoid excessive overheads on one hand, while maintaining the capability to adapt quickly to changing performance bottleneck shifts.

The success of the above algorithm depends on the accuracy of the performance predictions of the data preprocessing stage. To this end, for a specific training sample size, we build a piece-wise linear regression model that takes the number of threads as input and predicts the execution time of processing one training sample. We build a portfolio of models, each of which corresponds to a training sample size. During runtime, if the sample size does not have a corresponding model in the portfolio, we choose the model whose sample size is closest to the one considered.

Table 1: Notation used in performance models.

Notation	Metric	Description
N		number of compute nodes
M		number of GPUs in one compute node
Mem		storage space on each compute node
D		training dataset, comprising $\|D\|$ data samples
S		size of D
s_i		size of data sample d_i ($d_i \in D$)
I		number of iterations per epoch
$T_l(\alpha)$	MB/s	local memory read throughput (α read threads)
$T_r(\beta)$	MB/s	inter-node read throughput (β read threads)
$T_{PFS}(\gamma)$	MB/s	read throughput of remote PFS (γ read threads)

Note that the performance modeling approach mentioned above is architecture-dependent and data sample-dependent. This means that for different training environments (different hardware, types of preprocessing and sample sizes) we need to adjust the performance modeling. However, in practice, the same HPC machines, types of preprocessing and sample sizes are reused across many training instances of the same or different DNN models. Therefore, the cost of constructing performance model is amortized.

4.2 Coordinated Data Loading / Preprocessing

Given a fixed number of threads for the data preprocessing stage, the remaining CPU threads of the same node will be assigned for the data loading stage that serves all GPUs of that node.

Discriminating data load overheads between co-located GPUs. Current state of art efforts [8, 19, 26, 31, 44] serve all GPUs equally using a pool of threads reserved for the data loading stage. However, this is sub-optimal because the GPUs that trigger higher data loading overheads should be served using more threads, such that they will not become stragglers. To address this issue, Lobster proposes to maintain a separate request queue for each GPU, each of which can be assigned a different number of threads such as to achieve load balancing. Note that the data loading requests are placed in the queue of each GPU based on deterministic prefetching while considering the reuse distance. These details will be discussed later (Section 4.4).

Thread assignment. Given the requests in all GPU queues of a node that correspond to future training iterations, Lobster checks if there is a GPU that is predicted to become a straggler due to data loading. The prediction is based on performance modeling, as detailed in Section 4.3. Assignments are then made by using the heuristic detailed in Section 4.4. When a GPU is not predicted to become a straggler, the number of threads assigned to the request queue is proportional to the size of the queue.

4.3 Performance Model

We introduce a holistic performance model that bridges the thread management strategy for data loading and preprocessing (discussed above) with distributed caching. Table 1 summarizes the major notations used by our model.

Let s_i be the size of sample d_i in the training dataset D, which thus has a total size of $S = \sum_{0 \le i < |D|} s_i$; N be the number of nodes and M the number of GPUs per node (for a total of $N \times M$ GPUs); Mem be the host memory size allocated for caching; and $|B|$ be the mini-batch size. If $S > Mem$, then the training data cannot be fully cached on a single node; if $S > N \times Mem$, it cannot be fully cached across all nodes. One epoch consists of $I = \left\lceil \frac{|D|}{|B| \times M} \right\rceil$ iterations, or $I = \left\lfloor \frac{|D|}{|B| \times M} \right\rfloor$ iterations if we discard the last (potentially partial) iteration. At iteration h ($0 \le h < I$), each GPU G_i^j (where i is the node ID and j is the GPU ID), processes its own mini-batch $B^{h,i,j}$.

Overall, the GPUs need to read a collection $B^h = \bigcup_{i \in N, j \in M} B^{h,i,j}$ of training samples concurrently. Three scenarios can arise when reading training sample d_k with size s_k on node n_i:

(1) d_k is present in the local cache of node n_i, in which case the data loading duration is $\frac{s_k}{T_l(\alpha)}$, where $T_l(\alpha)$ is the local cache read throughput with α concurrent I/O threads.
(2) d_k is present in the remote cache of another node, in which case the data loading duration is $\frac{s_k}{T_r(\beta)}$, where $T_r(\beta)$ is the remote cache read throughput of a single I/O with β concurrent threads.
(3) d_k is present on the remote storage repository (parallel file system), in which case the data loading duration is $\frac{s_k}{T_{PFS}(\gamma)}$, where $T_{PFS}(\gamma)$ is the PFS read throughput of a single I/O thread with γ concurrent I/O threads (for simplicity, we assume $T_{PFS}(\gamma)$ to be globally stable on the average across the compute nodes).

Assume $B_{HL}^{h,i,j}$ and $B_{HR}^{h,i,j}$ represent the training samples that cause cache hits on node n_i's local cache and on the cache of remote nodes respectively, while $B_M^{h,i,j}$ represents training samples that cause cache misses and need to be fetched from the PFS. We have $B^{h,i,j} = B_{HL}^{h,i,j} \cup B_{HR}^{h,i,j} \cup B_M^{h,i,j}$. Assuming $T_L(n_i, B^{h,i,j})$ represents the duration of loading $B^{h,i,j}$ for GPU G_i^j, we have:

$$T_L(n_i, B^{h,i,j}) = \frac{\sum_{d_k \in B_{HL}^{h,i,j}} s_k}{\alpha_{i,j} \times T_l(\alpha_{i,j})} + \frac{\sum_{d_k \in B_{HR}^{h,i,j}} s_k}{\beta_{i,j} \times T_r(\beta_{i,j})} + \frac{\sum_{d_k \in B_M^{h,i,j}} s_k}{\gamma_{i,j} \times T_{PFS}(\gamma i, j)} \tag{1}$$

In Equation 1, $\alpha_{i,j}$, $\beta_{i,j}$ and $\gamma_{i,j}$ are the initial number of data loading threads allocated for each scenario for each GPU G_i^j (as discussed in Section 4.2). Given the mini-batch $B^{h,i,j}$, we denote its data preprocessing time $T_P(n_i, B^{h,i,j})$. We assume the duration of the training stage T_{train} is constant. Thus, in order to minimize the performance bottleneck introduced by the data loading and preprocessing stages for GPU G_i^j during iteration h, we need to minimize the following expression:

$$\min \left| T_L(n_i, B^{h,i,j}) + T_P(n_i, B^{h,i,j}) - T_{train} \right| \tag{2}$$

However, our overall goal is to minimize the performance bottleneck introduced by the data loading and preprocessing of all M GPUs of the compute node during iteration h. Assuming $T_{max}^{h,i}$ and $T_{min}^{h,i}$ are the maximum and minimum $T^{h,i,j}$ at iteration h across all M GPUs (where $T^{h,i,j}$ is the execution time of the h^{th} iteration for the j^{th} GPU on the node i), we can achieve this goal by minimizing the gap between $T_{max}^{h,i}$ and $T_{min}^{h,i}$ as follows:

Algorithm 1: Heuristic algorithm to find the near-optimal number of data loading threads for a given mini-batch.

Input: ℓ_{max}: Max number of threads for data loading: T_L
ℓ_{min}: Min number of threads for data loading: 0
B: Batch of data samples to be prefetched
G: List of co-located GPUs on a node; $|G| = M$
τ: Threshold to constrain load balance
Output: A list of assigned data loading threads: L_{final}

```
L_th = initial allocation of data loading threads
W = ∅
for i = 0 to |G| do
    T_dif = TimeDifference(G_i, B_i, L_th^i)
    if |T_dif| ≥ τ then
        L_th^opt = L_th^i // Update threads number
        T_dif^min = T_dif // Save minimal time difference
        while |T_dif| ≥ τ do
            W = W ⊕ T_dif
            if |W| > T_L and IsConsistent(W) then
                break
            if T_dif < 0 then
                ℓ_min = L_th^i
            else
                ℓ_max = L_th^i
            L_th^new = ℓ_min + (ℓ_max − ℓ_min)/2
            T_dif = TimeDifference(G_i, B_i, L_th^new)
            if T_dif^min ≤ T_dif then
                L_th^opt = L_th^new
                T_dif^min = T_dif // Update time difference
        W = ∅
    ℓ_min = 0, ℓ_max = T_L // Recover the values
    L_final = L_final ⊕ L_th^opt
return L_final
```

$$\min \left| T_{max}^{h,i} - T_{min}^{h,i} \right| \tag{3}$$

Unfortunately, the solution for Equations 2 and 3 is an optimization problem that can be solved using Integer Linear Programming (ILP), which is known to be NP-complete [38]. Even if this were feasible for a single iteration h, we have to consider that we have a total of $N \times M$ GPUs and a large number I of iterations. Thus, an exact solution to this optimization problem is not tractable.

4.4 Heuristic Strategy

To make the optimization problem introduced above tractable, we propose a heuristic strategy that works in two phases: (1) determine the number of data loading threads for each GPU; (2) determine an efficient eviction strategy for deterministic prefetching to avoid cache misses due to evicting samples with small reuse distance. Note that (2) influences the scenarios applicable for the training samples (node-local cache vs. remote cache vs. PFS), therefore there is a close connection between (1) and (2).

Thread assignment in case of predicted stragglers. Given a set of mini-batches to be prefetched by the GPUs co-located on a node, Lobster determines a near-optimal assignment of data loading threads by using a greedy algorithm (detailed in Algorithm 1) that aims to satisfy the goals formulated in Equations 2 and 3.

The initial allocation L_{th} of data loading threads to each co-located GPU is proportional to the number of pending requests in the data loading queue. We then calculate the difference between the duration of data loading + preprocessing and that of the training stage (Line 4) by using Equations 1 and 2. If this difference is greater than a threshold τ (which can be fine-tuned as needed to prune the search space), then we employ a binary search to explore the search space until we converge to a near-optimal solution that minimizes the gap between T_{max} and T_{min}.

To cover the case when the greedy algorithm does not converge, we introduce an array $\mathcal{W}$ whose length is T_L, the maximum number of data loading threads that can be used by a node. The array records T_{dif} calculated in the prior iterations (Line 9). When the array is fully populated, then we stop the search and choose the solution that has the minimum T_{dif} among all those recorded in $\mathcal{W}$.

Eviction Policy based on Reuse Distance and Deterministic Prefetching. It is important to note that the determinism of the prefetching pattern of one node is a *global* property: it is known to *all* other nodes (e.g. by fixing the pseudorandom number generator seed of each node such that it is a function of a fixed seed and the node id). Thus, we can determine, at each moment during training, two parameters: (1) how many times each training sample will be reused by all GPUs until the end of training; (2) the minimum reuse distance of each training sample across all GPUs. To obtain these parameters efficiently, we maintain a list of future accesses for each training sample. Each entry in the list records the GPU and iteration number during which the training sample needs to be accessed for the reminder of the training. Based on this list, we apply two sub-policies:

Reuse count policy. If during training the number of accesses to a training sample reaches the reuse count for a node, then the sample is evicted from the node-local cache—unless no other node in the group holds a copy, as eviction would then force the other nodes to perform expensive I/O operations to re-prefetch the training sample from the remote storage repository.

Reuse distance policy. Let I be the number of iterations in an epoch, h the current iteration, and B^h the set of mini-batches accessed by all co-located GPUs on a node. Then after iteration h has finished, we can check the next reuse distance of each training sample $d_k \in B^h$. If the next reuse distance is larger than $2 \times I - h$, then the training sample will not be accessed by any GPUs on the node during the next epoch. In this case, the training sample can be considered as being reused far enough in the future to justify eviction in order to make room for more prefetches.

Coordination with prefetching. Thanks to the two eviction policies mentioned above, any spare capacity in the node-local cache can be used for prefetching more training samples. However, if the training samples can be prefetched faster than they are consumed, the spare capacity will be quickly filled. In this case, we can evict the training samples with the largest reuse distance, while prioritizing the prefetches with the nearest reuse distance.

4.5 Implementation Details

Lobster consists of two components: one is used in offline fashion to construct piece-wise linear regression models for the preprocessing stage and to pre-compute an efficient thread management plan combined with an efficient prefetching/eviction plan based on the reuse distance. The planning phase is based on a simulator proposed by [8], which was extended to: (1) decide the number of data preprocessing threads; (2) decide the number of data loading threads; (3) adding the cache eviction algorithm, and (4) add coordination logic between data preprocessing and data loading.

The other component is an online runtime implemented in C++ and built on the top of DALI 2.0 [44]. It is designed to interpret the plan generated by the offline component, and to enforce the thread management and data prefetching as planned. A key part of the online runtime is the distribution manager, responsible to handle the distributed operations across the compute nodes using MPI. These operations provide locally cached training samples to and request training samples from the remote compute nodes.

5 EVALUATION

We evaluate the performance of Lobster in two data-parallel scenarios: (1) single node with multiple GPUs and (2) multiple nodes, each with multiple GPUs. In each case, we compare Lobster with three baseline approaches in terms of I/O performance, memory cache efficiency, and end-to-end training runtime. Our evaluation aims to answer four questions:

- Does Lobster have better I/O performance than the baselines? (Section 5.2)
- Does Lobster address the load imbalance problem? (Section 5.3)
- Does Lobster influence end-to-end training performance compared with baselines? (Section 5.4)
- Does each component of Lobster contribute to the overall improvement? (Section 5.6)

5.1 Experimental Setup

We evaluate Lobster with six representative DNN models that are frequently used as training benchmarks (ResNet50 [12], ResNet32 [12], ShuffleNet [41], AlexNet [18], SquenceNet [11], VGG11 [37]) and, for each model, two datasets, IMAGENet-1K and ImageNet-22K. We use PyTorch 1.8 with NCCL2 for all evaluations. At the beginning of DNN model training, the training datasets are stored on a Lustre parallel file system mount point.

Baselines. We compare Lobster with three baseline approaches:

- PyTorch I/O [31]: The built-in PyTorch DataLoader using a constant number of threads for data loading and another constant number of threads for preprocessing.
- DALI [44]: A widely used NVIDIA library for DNN training I/O. DALI uses three threads for data loading by default and leaves other threads for preprocessing.
- NoPFS [8]: A state-of-the-art approach that implements deterministic prefetching that is combined with PyTorch. The thread management for NoPFS is the same as that with PyTorch I/O.

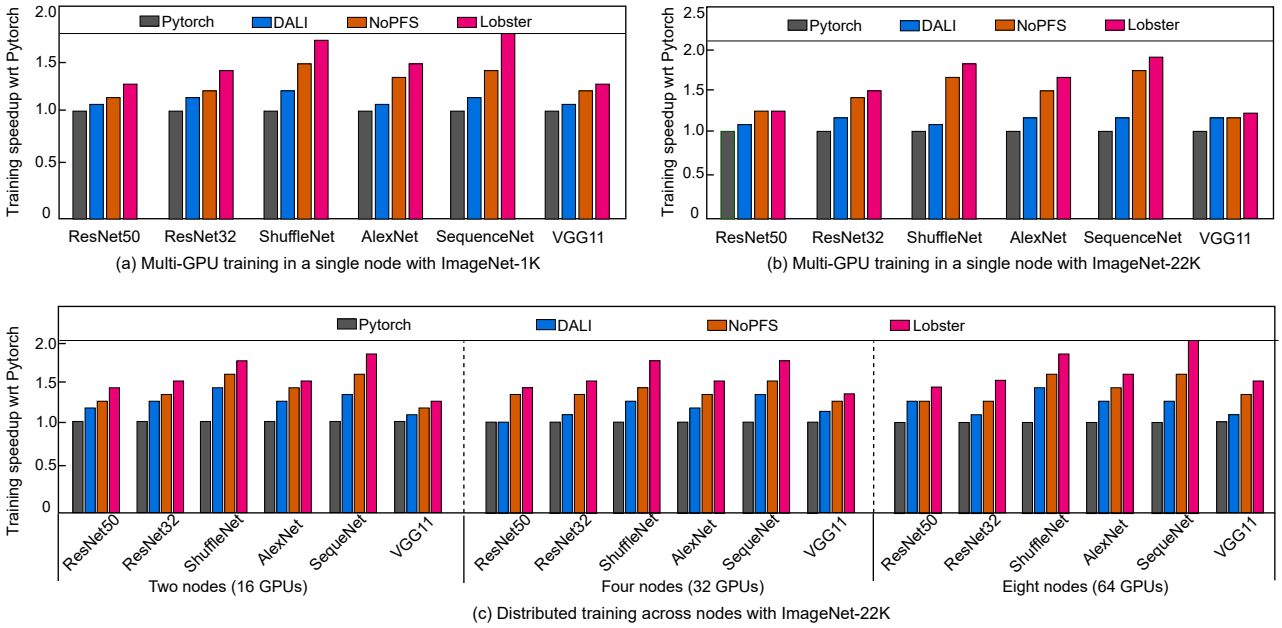

Figure 7: Comparison between Lobster and the baselines for multi-GPU training on a single node and distributed training across multiple nodes.

Hardware. We performed experiments on Argonne's ThetaGPU HPC machine, which is specifically optimized for training DNNs at scale. It comprises 24 NVIDIA DGX A100 nodes, each of which is equipped with eight NVIDIA A100 Tensor-Core GPUs, two AMD Rome CPUs, 1 TB of DDR4 memory and 320 GB GPU memory. This amounts to a total of 24 TB DDR4 and 7.6 TB GPU memory. We use 40 GB DDR4 memory as node-local cache on each node. If the cache is large, all samples are placed locally without causing I/O. But typically, a small portion of DDR4 memory is used as cache. The nodes are interconnected with 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology. The external storage provided by a Lustre parallel file system deployment provides an aggregate 250 GB/s bandwidth, mounted using POSIX.

Datasets. We use two widely used datasets with different sizes.

- ImageNet-1K [36]: This dataset, widely used to evaluate DNNs for image classification, consists of 1.28 million training images and 50,000 validation images, each image assigned to one of 1000 classes. Its total size is 135 GB.
- ImageNet-22K [35]: A representative larger dataset widely used to pre-train DNNs. This dataset consists of 14,197,103 training images and 7000 validation images, most with an image size of between 10 KB and 50 KB, and each assigned to one of 21,841 classes. The total dataset size is 1.3 TB.

5.2 I/O Performance

Single-Node Multi-GPU Data-Parallel Training. We run Lobster on a single node with eight GPUs. Figure 7(a) and (b) show the results for ImageNet-1K and ImageNet-22K, respectively. We observe that:

(a) Lobster is 1.6× and 1.8× faster than PyTorch DataLoader using ImageNet-1K and ImageNet-22K, respectively. This performance improvement results mainly from the alleviation of I/O load imbalance between GPUs. The performance improvement is especially large in the case of ImageNet-22K (the larger dataset).

(b) Lobster is 1.7× faster than DALI, for two reasons: (1) DALI lacks fine-grained thread-level optimizations for the training pipeline, while Lobster can flexibly allocate CPU threads to coordinate data loading and pre-processing; and (2) Lobster is NUMA-aware, and co-locates data loading and preprocessing threads.

(c) Lobster is 1.2× faster than NoPFS. This is due to the cache eviction based on reuse distance, which complements deterministic prefetching. Specifically, NoPFS evicts the training samples to accommodate the training samples to be prefetched for the next iteration, while our approach is able to prefetch more training samples thanks to its eviction policies, which translates to higher cache hit rates and better performance.

Multi-Node Distributed Data-Parallel Training. We evaluate Lobster on eight nodes when using all eight GPUs available on each node. We see in Figure 7(c) that for ImageNet-22K Lobster is 2.0×, 1.4×, and 1.2× faster than PyTorch DataLoader, DALI, and NoPFS, respectively. Compared with the single-node evaluation, Lobster makes better use of the distributed cache across nodes, leading to a larger performance improvement.

Scalability of Data-Parallel Training. We evaluate Lobster using a variable number of nodes and with different datasets. For ImageNet-1K, we use a single node whose memory cache is smaller than the dataset size, while for ImageNet-22K, we use multiple nodes whose aggregated memory cache across the nodes is smaller than the dataset size (but larger than the size of ImageNet-1K).

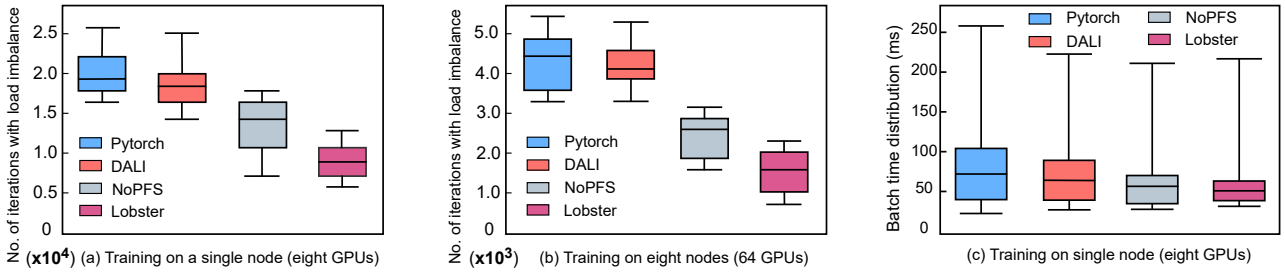

Figure 8: The number of iterations with load imbalance and the distribution of batch time. We use ResNet50 with ImageNet-1K.

Figure 7 shows the results with respect to PyTorch Dataloader: (a) Lobster scales well for ImageNet-1K. Furthermore, when training with ImageNet-22K, compared with PyTorch Dataloader, Lobster has a speedup of 1.53× on average (up to 1.9×); (b) with different system scales on a single node and multiple nodes, Lobster consistently shows a significant speedup (1.2×–2.0×).

5.3 Reduction of Load Imbalance

To evaluate Lobster's effectiveness in reducing data load imbalance, we count the number of iterations with load imbalance across GPUs in each epoch. We use ResNet50 with ImageNet-22K as the training dataset. Figure 8 depicts the results.

Single-Node Multi-GPU Data-Parallel Training. We train the model for 50 epochs, each of 55,457 iterations. We depict in Figure 8(a) the results for all epochs. Compared with PyTorch, DALI, and NoPFS, Lobster reduces the iterations with load imbalance by 31.4%, 16.4%, and 7.9% respectively on average. WithLobster, only 17.5% of all training iterations exhibit load imbalance.

Multi-Node Distributed Data-Parallel Training. We fix the number of iterations per epoch at 6932 and train ResNet-50 for 50 epochs. Figure 8(b) shows the results for all epochs. The number of iterations is smaller than when training on a single node because we use more GPUs. Compared with PyTorch DataLoader, DALI, and NoPFS, Lobster reduces the number of iterations with load imbalance by 35.2%, 25.8%, and 9.7% respectively. Overall, with Lobster, only 22.8% of all training iterations still exhibit load imbalance.

Lobster is able to reduce the number of iterations with load imbalance more effectively thanks to the coordination between the data loading and preprocessing stages, which redirects more threads for data loading when the GPUs are bottlenecked by it.

Batch time distribution. Figure 8(c) presents the batch time distribution when training ResNet50 with ImageNet-1K on one node (8 GPUs), showing successful mitigation of performance degradation brought by the load imbalance across GPUs. With Lobster, there is less variance in per-batch time (iteration duration) than the baselines. Lobster's batch time is also shorter than other methods'. Figures 8 demonstrates a key performance advantage of Lobster: reducing the batch time where the data loading is slow due to load imbalance across GPUs.

5.4 End-to-End Training

Lobster does not change the randomness of data accessing during the distributed training. The techniques in Lobster do not influence the DNN model's accuracy. To demonstrate this, we train ResNet50 to convergence for ImageNet-1K on eight nodes (64 GPUs) with both Pytorch DataLoader and Lobster. Figure 9 shows the accuracy curves. We see that the two methods have similar learning curves, although with some slight variation due to different random seeds for network parameters. In both cases, training converges to the target accuracy of 76.0% in around 40 epochs. Nevertheless, as Figure 7 indicates training with Lobster is up to 1.4× faster than with PyTorch DataLoader. As a consequence, using Lobster for data loading achieves an overall shorter training time.

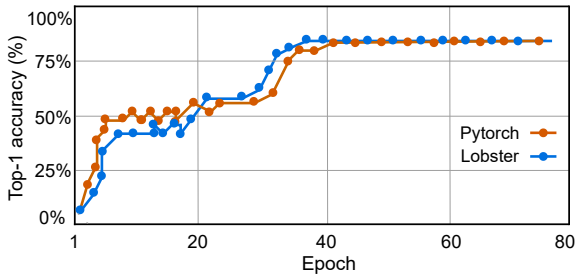

Figure 9: Training accuracy curve for training ResNet50 on ImageNet-1K using eight nodes (64 GPUs) with default ResNet50 hyperparameter settings.

5.5 Resource Utilization

Memory cache hit ratio. We measure the cache hit ratio of the memory cache during the whole training process. We use one node with eight GPUs and ImageNet-1K. Lobster has higher cache hit ratio than the baselines. On average, the cache hit ratio with Lobster is 63.2%, while it is 24.5%, 32.6%, and 48.9% with PyTorch DataLoader, DALI, and NoPFS respectively. The higher cache hit ratio demonstrates the effectiveness of cache eviction as a complement for deterministic prefetching. NoPFS has higher cache hit ratio than PyTorch DataLoader and DALI, because of its efficient distributed cache with deterministic prefetching. However, its cache hit ratio is lower than Lobster because of a simpler cache eviction policy.

GPU utilization. We measure average GPU utilization during the whole training process. We use one node with eight GPUs and ImageNet-1K, and fix the number of epochs at 50. As can be observed in Figure 10, Lobster has higher GPU utilization than the baselines: 76.1% vs. 52.3%, 57.5%, and 72.4% for PyTorch DataLoader,

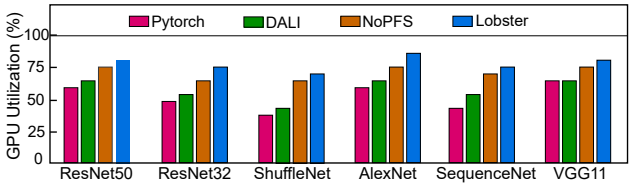

Figure 10: GPU utilization when training ResNet50 on ImageNet-1K using one node (eight GPUs). X-axis represents different DNNs used for testing and Y-axis is the GPU utilization.

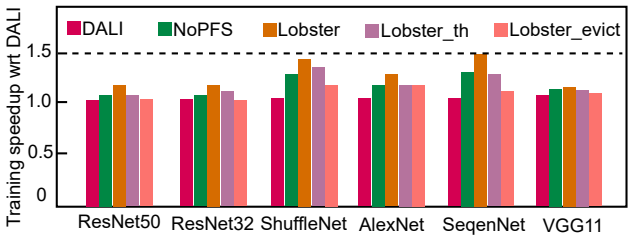

Figure 11: Ablation study of Lobster when training ResNet50 on ImageNet-1K using one node (eight GPUs). Y-axis is the training time speedup compared with DALI.

DALI, and NoPFS respectively. The higher GPU utilization of Lobster demonstrates the effectiveness of addressing load imbalance across GPUs.

5.6 Ablation Study

We next evaluate the individual impacts of Lobster's thread management and its cache eviction policies. To this end, *Lobster_th* includes thread management but excludes cache eviction based on reuse distance, while *Lobster_evict* does the precise opposite. We show in Figure 11 results for ImageNet-1K on a single node (eight GPUs).

Thread management. We see that: (1) the thread management optimization contributes more to the training performance improvement than the cache eviction policy; (2) the thread management improves the training performance by up to 1.4× (1.3× on average), compared with DALI.

Cache eviction policy. We also see that the cache eviction policy based on reuse distance (1) leads to 15% higher performance than DALI, on average, and (2) is more helpful for small models (e.g., ShuffleNet, SequenceNet), for which the duration of the training stage is smaller compared with the larger models (and thereby less likely to become a performance bottleneck).

6 RELATED WORK

Optimizing DNN training time. Solutions proposed for reducing DNN training times include specialized hardware [3, 30], parallel training [17, 23, 32], GPU memory optimizations [13], lowering communication overhead [9], and operator optimizations [33, 34]. New hardware systems like NVIDIA's Magnum IO [3] provide high-throughput storage solutions to reduce data loading overheads, but cannot help when training DNNs in distributed environments with complex storage hierarchies. Model batching [29] addresses data loading costs when running multiple DNNs on a single node. OneAccess [16] is a preliminary study that makes a strong case for storing pre-processed data across epochs to reduce the data preprocessing overhead. However such an approach precludes commonly used online data preprocessing techniques, which can affect model convergence during training. None of these approaches address, as does Lobster, data loading overheads resulting from load imbalances across GPUs. Lobster balances loads between GPUs and avoids bursty data loading such that the data loading does not become a performance bottleneck.

Data caching for distributed DNN training. Quiver [19] uses SSD as caches to avoid slow data loading from remote storage. Cerebro [27] partitions the dataset across nodes in a cluster. Instead of shuffling data, Cerebro moves the models from one node to others. However, when training DNNs with a single node, using Cerebro cannot bring performance improvement. DeepIO [43] uses a partitioned caching technique for distributed training. DeepIO heavily relies RDMA for high performance I/O. DIESEL [39] deploys a task-grained distributed cache across nodes for multiple DNN training tasks. DIESEL introduces metadata snapshot mechanisms for each training dataset, and mainly focuses on optimizing metadata processing during the DNN training. MinIO [26] reduces the amount of disk I/O for training on a single node and multiple nodes. MinIO does not provide the fine-grained cache strategy like Lobster. For MinIO, once data samples are cached, they are never evicted out of the cache. NoPFS [8] introduces a performance model that can leverage the storage hierarchy for the data caching. But NoPFS cannot immediately evict data samples with long reuse distances out of memory. Lobster addresses the I/O bottleneck by providing the thread management for different stages in the training pipeline. Furthermore, Lobster leverages the knowledge on the reuse distance of data samples to make the best use of the memory cache.

7 CONCLUSIONS

Data loading is becoming a major performance bottleneck in distributed DNN training. Prior studies of data loading performance for distributed DNN training have conducted neither a holistic analysis of all training pipeline stages not a fine-grained analysis of the load of individual GPUs, two areas that present opportunities for further optimization. To fill this gap, we have proposed Lobster, a data loading runtime that exploits several observations related to load imbalance, performance bottlenecks in various stages of the training pipeline, and the reuse distance of training samples to propose a new flexible thread management strategy and cache eviction policy that complements deterministic prefetching. These methods allow Lobster to consistently outperform the state-of-art PyTorch I/O, DALI, and NoPFS systems by 1.3–2.0×.

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. 2016. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467* (2016).

[2] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. 2021. Accelerating recommendation system training by leveraging popular choices. *Proceedings of the VLDB Endowment* 15, 1 (2021), 127–140.

[3] Idan Burstein. 2021. Nvidia Data Center Processing Unit (DPU) Architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–20.

[4] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K. Panda. 2020. NV-group: Link-efficient reduction for distributed deep learning on modern dense GPU systems. In *34th ACM International Conference on Supercomputing*. Virtual, 1–12.

[5] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. 2008. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*.

[6] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. 2019. Adaptive Neural Network-Based Approximation to Accelerate Eulerian Fluid Simulation. In *International Conference for High Performance Computing, Performance Measurement, Modeling and Tools (SC)*.

[7] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. 2020. Smart-PGSim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation. In *International Conference for High Performance Computing, Performance Measurement, Modeling and Tools* .

[8] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. 2021. Clairvoyant prefetching for distributed machine learning I/O. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[9] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*. IEEE, 1–8.

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

[11] A Handa, V Patraucean, V Badrinarayanan, S Stent, and R Cipolla. 2015. SceneNet: Understanding real world indoor scenes with synthetic data. *arXiv preprint arXiv:1511.07041* (2015).

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*. 770–778.

[13] Chien-Chin Huang, Gu Jin, and Jinyang Li. 2020. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1341–1355.

[14] 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 *4th Conference on Machine Learning and Systems*. Virtual.

[15] 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. In *22nd ACM International Conference on Multimedia*. 675–678.

[16] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. 2019. The case for unifying data loading in machine learning clusters. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[17] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. 2019. Parallax: Sparsity-aware data parallel training of deep neural networks. In *15th EuroSys Conference*. 1–15.

[18] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[19] Abhishek Vijaya Kumar and Muthian Sivathanu. 2020. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies*. 283–296.

[20] 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 *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 649–660.

[21] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. 2021. Refurbish Your Training Data: Reusing Partially Augmented Samples for Faster Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 537–550.

[22] Dong Li, Bronis de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. 2010. Hybrid MPI/OpenMP power-aware computing. In *International Parallel and Distributed Processing Symposium*.

[23] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. 2020. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704* (2020).

[24] Jie Liu, Bogdan Nicolae, Dong Li, Justin M Wozniak, Tekin Bicer, Zhengchun Liu, and Ian Foster. 2022. Large Scale Caching and Streaming of Training Data for Online Deep Learning. In *Proceedings of the 12th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures*. 19–26.

[25] Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason Haga, and Joao Paulo. 2021. The Case for Storage Optimization Decoupling in Deep Learning Frameworks. In *IEEE International Conference on Cluster Computing*. IEEE, 649–656.

[26] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. 2020. Analyzing and mitigating data stalls in DNN training. *arXiv preprint arXiv:2007.06775* (2020).

[27] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2159–2173.

[28] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: Generalized pipeline parallelism for DNN Training. In *27th ACM Symposium on Operating Systems Principles*. Huntsville, Canada, 1–15.

[29] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. 2018. Accelerating model search with model batching. In *1st Conference on Systems and Machine Learning (SysML), SysML*, Vol. 18.

[30] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. 2020. TrainBox: An Extreme-Scale Neural Network Training Server Architecture by Systematically Balancing Operations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 825–838.

[31] 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. *Advances in Neural Information Processing Systems* 32 (2019), 8026–8037.

[32] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. 2014. Parallel training of deep neural networks with natural gradient and parameter averaging. *arXiv preprint arXiv:1410.7455* (2014).

[33] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning. *arXiv preprint arXiv:2104.07857* (2021).

[34] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. 2021. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840* (2021).

[35] Tal Ridnik, Emanuel Ben-Baruch, Asaf Noy, and Lihi Zelnik-Manor. 2021. Imagenet-21k pretraining for the masses. *arXiv preprint arXiv:2104.10972* (2021).

[36] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. 2015. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision* 115, 3 (2015), 211–252.

[37] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[38] Vasilis Sourlas, Lazaros Gkatzikis, Paris Flegkas, and Leandros Tassiulas. 2013. Distributed cache management in information-centric networks. *IEEE Transactions on Network and Service Management* 10, 3 (2013), 286–299.

[39] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. 2020. DIESEL: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing*. 1–11.

[40] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. MD-HM: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*. 215–226.

[41] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition*. 6848–6856.

[42] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. 2021. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *arXiv preprint arXiv:2108.09373* (2021).

[43] 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 *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 145–156.

[44] Mahdi Zolnouri, Xinlin Li, and Vahid Partovi Nia. 2020. Importance of data loading pipeline in training DNNs. *arXiv preprint arXiv:2005.02130* (2020).