Scalability Analysis and Optimization for Large-Scale Deep Learning

Sarunya Pumma

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Wu-chun Feng, Chair
Ali R. Butt
Dongyoon Lee
B. Aditya Prakash
Pavan Balaji

December 18, 2019
Blacksburg, Virginia

Keywords: Scalable Deep Learning, Data Movement Investigation, Parallel I/O Optimization, Straggler Processes, Computational Imbalance, Resource Contention

Scalability Analysis and Optimization for Large-Scale Deep Learning

Sarunya Pumma

(ABSTRACT)

Despite its growing importance, scalable deep learning (DL) remains a difficult challenge. Scalability of large-scale DL is constrained by many factors, including those deriving from *data movement* and *data processing*. DL frameworks rely on large volumes of data to be fed to the computation engines for processing. However, current hardware trends showcase that data movement is already one of the slowest components in modern high performance computing systems, and this gap is only going to increase in the future. This includes data movement needed from the filesystem, within the network subsystem, and even within the node itself, all of which limit the scalability of DL frameworks on large systems. Even after data is moved to the computational units, managing this data is not easy. Modern DL frameworks use multiple components—such as graph scheduling, neural network training, gradient synchronization, and input pipeline processing—to process this data in an asynchronous uncoordinated manner, which results in straggler processes and consequently computational imbalance, further limiting scalability. This thesis studies a subset of the large body of data movement and data processing challenges that exist in modern DL frameworks.

For the first study, we investigate file I/O constraints that limit the scalability of large-scale DL. We first analyze the Caffe DL framework with Lightning Memory-Mapped Database (LMDB), one of the most widely used file I/O subsystems in DL frameworks, to understand

the causes of file I/O inefficiencies. Based on our analysis, we propose LMDBIO—an optimized I/O plugin for scalable DL that addresses the various shortcomings in existing file I/O for DL. Our experimental results show that LMDBIO significantly outperforms LMDB in all cases and improves overall application performance by up to 65-fold on 9,216 CPUs of the Blues and Bebop supercomputers at Argonne National Laboratory.

Our second study deals with the computational imbalance problem in data processing. For most DL systems, the simultaneous and asynchronous execution of multiple data-processing components on shared hardware resources causes these components to contend with one another, leading to severe computational imbalance and degraded scalability. We propose various novel optimizations that minimize resource contention and improve performance by up to 35% for training various neural networks on 24,576 GPUs of the Summit supercomputer at Oak Ridge National Laboratory—the world's largest supercomputer at the time of writing of this thesis.

Scalability Analysis and Optimization for Large-Scale Deep Learning

Sarunya Pumma

(GENERAL AUDIENCE ABSTRACT)

Deep learning is a method for computers to automatically extract complex patterns and trends from large volumes of data. It is a popular methodology that we use every day when we talk to Apple Siri or Google Assistant, when we use self-driving cars, or even when we witnessed IBM Watson be crowned as the champion of Jeopardy! While deep learning is integrated into our everyday life, it is a complex problem that has gotten the attention of many researchers.

Executing deep learning is a highly computationally intensive problem. On traditional computers, such as a generic laptop or desktop machine, the computation for large deep learning problems can take years or decades to complete. Consequently, supercomputers, which are machines with massive computational capability, are leveraged for deep learning workloads. The world's fastest supercomputer today, for example, is capable of performing almost 200 quadrillion floating point operations every second. While that is impressive, for large problems, unfortunately, even the fastest supercomputers today are not fast enough. The problem is not that they do not have enough computational capability, but that deep learning problems inherently rely on a lot of data—the entire concept of deep learning centers around the fact that the computer would study a huge volume of data and draw trends from it. Moving and processing this data, unfortunately, is much slower than the computation itself and with

the current hardware trends it is not expected to get much faster in the future.

This thesis aims at making deep learning executions on large supercomputers faster. Specifically, it looks at two pieces associated with managing data: (1) data reading—how to quickly read large amounts of data from storage, and (2) computational imbalance—how to ensure that the different processors on the supercomputer are not waiting for each other and thus wasting time. We first analyze each performance problem to identify the root cause of it. Then, based on the analysis, we propose several novel techniques to solve the problem. With our optimizations, we are able to significantly improve the performance of deep learning execution on a number of supercomputers, including Blues and Bebop at Argonne National Laboratory, and Summit—the world's fastest supercomputer—at Oak Ridge National Laboratory.

Dedication

To my beloved family, Fizzbuzz, and Foobar.

Acknowledgments

First, I would like to thank my Ph.D. advisor, Dr. Wu-chun Feng, for giving me the one-of-a-kind opportunity to work and grow under the roof of the Synergy Lab. Throughout the course of my Ph.D. study, I have learned tremendously from his incredible intellect, horizons and work ethics. I truly appreciate his constructive criticism, comments, and advice that have helped me successfully accomplish my Ph.D. journey.

I am incredibly fortunate to have a chance to work under the supervision of Dr. Pavan Balaji, the most talented and kindest person I have ever met. Not only has he taught me to do research but also to love and embrace it. I deeply appreciate his constant encouragement and support that have brought out the potential I never thought I had. Thank you for never giving up on me and always motivating me to go the extra mile in every aspect of my life. Without his kind support and guidance, this Ph.D. would not have been possible.

I would like to express my sincere thanks to my advisory committee members, Dr. Ali Butt, Dr. Dongyoon Lee, and Dr. Aditya Prakash, for their precious time, comments and feedback on my thesis work. Thank you for being very kind to me.

I would also like to thank my amazing research mentors, Dr. Min Si, Dr. Daniele Buono, Dr. Fabio Checconi, and Dr. Xinyu Que. Their comments and feedback have been greatly valuable to my thesis as well as academic papers.

Moreover, I would like to give a big shout out to all my friends and supporters: Dr. Tiranee Achalakul (my former advisor), my Blacksburg family: Unchalisa, Gade, Panupon and the Charoenvisals, my lab mates, my room mates and everyone who called to check in, gave me

a pat on the back, and put a smile on my face every now and then. Thank you for filling my Ph.D. adventure with so much joy.

Lastly, I am extremely grateful to have unconditional and endless support, encouragement, and love from my family. Thank you for being there with me throughout every step of the way. I absolutely could not ask for a better family.

Contents

List of Figures

List of Tables

List of Abbreviations

CPU Central Processing Unit

DL Deep Learning

DNN Deep Neural Network

GPU Graphics Processing Unit

HPC High Performance Computing

I/O Input/Output

LMDB Lightning Memory-Mapped Database

Mmap Memory-Mapped File I/O

MPI Message Passing Interface

OS Operating System

Chapter 1

Introduction

Deep learning (DL) [91] is an emerging technology that is gaining prominence in a multitude of domains [18, 28, 45, 48, 64, 71, 93, 139, 154, 165] because of its ability to process unstructured input and to predict trends. Training of deep neural networks (DNNs) [60], a process where large volumes of input data are mined to find patterns and trends, usually involves high computational and memory complexity. These high resource requirements are growing rapidly as researchers are developing many DNNs [62, 69, 84, 90, 101, 145, 152, 153], with larger and more complex structures, to accurately process larger and more sophisticated data. To meet these complex resource demands, we note three broad trends in the community. First, researchers have targeted scalable high-performance computing as a mechanism to process data in parallel across multiple processors [25, 38, 46, 96, 117, 174]. Second, there has been a large influx of commercial hardware that is either tuned for or dedicated to DL systems. This hardware includes processors (e.g., NVIDIA GPUs [61, 119], Intel Xeon Phi [47], Google TPUs [53], Cerebras Systems Wafer Scale Engine—world's largest computer chip for artificial intelligence tasks [5]), high-speed networks (e.g., Mellanox InfiniBand [43, 116], Intel OmniPath [178]), and memory & storage technologies (e.g., Non-Volatile Memory Express or NVMe [175], Intel Optane memory [172]). Third, researchers have developed numerous algorithms to make DL more computationally efficient by allowing them to realize more algorithmic parallelism without losing inference accuracy. One of the very first breakthroughs in this area of research was shown in [181, 183] where the authors demonstrated parallelism

across 32,768 data samples (i.e., batch size) within each iteration with negligible loss in inference accuracy for ImageNet training. This area is evolving very rapidly. In fact, only about a year later, Osawa et al. [122] demonstrated using batch sizes of 131,072 for ImageNet training without losing accuracy. These batch size numbers are expected to go up dramatically in the next few years. Moreover, many parallel DL frameworks have been proposed in the past decade that incorporate the cited trends in usable software instantiations, including Caffe [3, 15, 68, 92], TensorFlow [6, 166], Theano [105, 161], Caffe2 [148], PyTorch [79], Microsoft Cognitive Toolkit [135], Apache MXNet [21], and Chainer [164]. Nevertheless, scalable DL remains a difficult problem to solve.

1.1 DL for HPC vs. HPC for DL

Large-scale DL is heavily intertwined with high performance computing (HPC), with each of them making substantial contributions in improving the other. DL has been utilized to support research and development in several domains including HPC. Similarly, breakthroughs in HPC are among the main contributors to the rapid growth of DL techniques. In light of this close relationship between DL and HPC, it is imperative that we distinguish two aspects of this relationship (i.e., using DL to improve HPC vs. using HPC to improve DL) and clarify which aspect this thesis focuses on.

DL for HPC: Enhancing the performance of a HPC system/application is nontrivial as the process usually involves high-dimensional performance metrics. Moreover, the relationship between these performance metrics cannot typically be represented using a simple mathematical model but would rather need a high-order polynomial or sometimes even a nonpolynomial function. Thus, DL is an appealing method for realizing the complex representation of the HPC system/application's performance metrics. DL has been used largely in three

broad areas in HPC. The first area is DL for automating performance tuning of the HPC systems/applications (or "autotuning"), for example, GPU parameter tuning [31, 49, 107], MPI runtime parameter tuning [127], and parallel application performance tuning [29, 44, 163]. With DL, the user of an HPC system/application can heuristically explore a large search space of performance-related parameters and derive close-to-optimal sets of parameters in a finite amount of time. The second area is DL for modeling complex HPC system/application performance, for example, parallel application performance prediction [146, 162], and HPC system's health prediction [36]. The third area is DL for HPC task scheduling [94, 120] or workload partitioning [57]. This is closely related to the second area in that DL is adopted to model the task scheduling/partitioning algorithm.

While using DL to enhance HPC workloads is an active and interesting area of research by itself, it is not within the scope of this thesis.

HPC for DL: Due to the rapid growth of data, complex DL methods and DNNs continue to be created to support fast and accurate data processing. This makes DL workloads prime candidates for running on HPC platforms where their complex resource demands can be easily satisfied. Scaling DL workloads to large HPC systems, however, remains a complex problem where one can experience significant performance loss if the massive computation resources are not carefully utilized and managed. Therefore, a significant amount of research has attempted to invent/improve/apply HPC techniques to allow DL workloads to perform well on large-scale HPC systems. Most of these efforts addressed network I/O optimizations [13, 180, 188]. Some recent research has also investigated other aspects of large-scale DL, such as data movement in file I/O [23, 77, 86, 88, 177, 180, 191] and between processors [22, 42, 124], and tradeoffs between performance and accuracy [55, 122, 181], some of which we have studied in this thesis.

This thesis focuses on ***HPC for DL*** where we seek to leverage supercomputers and HPC

techniques to accelerate DL workloads. We note that we have *not* modified the parallel DL algorithm. In this work, we focus on improving the *scalability* [17] of large-scale DL executions, i.e., their ability to handle a growing amount of data and/or resources. While there are dozens of factors that affect scalability, this thesis drills down into two significant aspects impacting the scalability of large-scale DL: data movement and data processing. Although improving a DL model's inference accuracy—the ability to correctly predict the output of an unseen input—is also important, it is *not* the main emphasis of this thesis. As such, we leverage well-known accuracy-improvement techniques showcased in [181, 183] in our work.

1.2 Challenges in Modern DL Frameworks

In this section, we present some of the challenges related to data movement and data processing in large-scale DL.

1.2.1 Data Movement Challenges

Here, we first describe ***three observations*** that motivate the data movement challenges in parallel DL.

First, contrary to popular belief, it is not true that all large-scale DL is compute bound. The specific challenges that a training model faces depend on the characteristics of each individual DL problem. We demonstrate the different challenges in large-scale DL through the following three real-world DL examples:

1. **Extreme weather detection [89]:** a semi-supervised bounding box prediction problem has two primary characteristics: (1) large batch sizes can be used in each training

iteration making the amount of data read in each iteration large; and (2) the computation is sparse on the data as the neural network is made up of sparse parameter layers. These two characteristics make this problem *file I/O* intensive.

2. **Lawrence Livermore National Laboratory's unsupervised image feature extraction [118]:** feature extraction is generally a highly complex problem that requires a complex neural network that consists of a large number of trainable parameters (15 billion parameters in this case) that can handle a variety of different features that each input image might have. Such characteristic makes the computation, and the amount of data read in each iteration, insignificant compared with the amount of communication needed in the parameter update step. Therefore, this kind of training is usually bound by *network I/O*.

3. **Data Science Bowl's tumor detection from CT scans [76]:** the size of the neural network, which is generally determined by a number of neurons, contributes to the amount of computation in the training. We note that the resolution of the input data and output data govern the number of neurons in the first layer and the last layer, respectively, while the number of neurons in the hidden layers is typically arbitrary. Thus, the input and output sizes have an impact on the size of the neural network to some degree. Therefore, the training that requires high-dimensional input data is generally bound by *computation*. CT scans would fall under this category.

The main takeaway from these examples is that DL problems vary widely. Some are compute bound while others are bound by data movement, which can be in the form of file I/O or network I/O or even data movement within the processor-memory subsystem.

Second, the technology trends today are more favorable to computation than they are to data movement. This enables compute-bound problems to take advantage of the increasing

processing speeds (through increasing core counts)—something that the other two classes of problems lack. For instance, processor speeds have improved by approximately 10-fold every ten years in the past few decades [137]. However, the same trend does not emerge in the filesystem and network I/O technologies.

Third, not all data movement is the same. The cost of data movement, with respect to access time and energy consumption, varies dramatically based on the distance that the data is moved. Moving data from main memory to registers takes significantly longer than performing a floating point operation, and this difference is only increasing with newer processors. Moreover, the lower a piece of data is in the memory hierarchy (Figure 1.1), the slower the data movement.

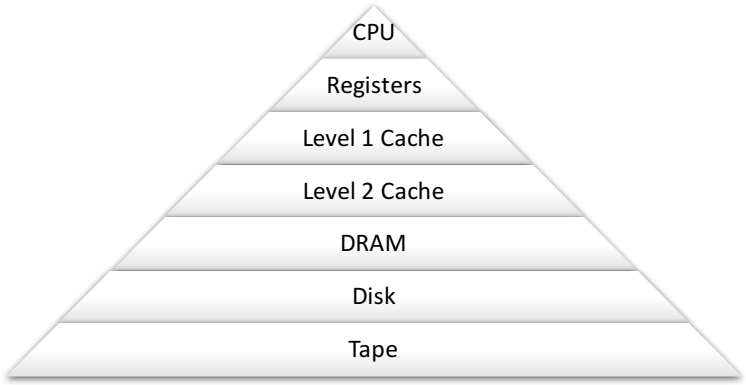

Figure 1.1: Memory hierarchy

Similar to access time, the energy cost to move data from memory to register is twice as much as the energy cost of a double-precision floating point computation [78]. The amount of energy required increases with the distance of data movement as demonstrated in Figure 1.2. The farther the data is moved, the more the energy used [104, 123, 143].

To summarize these observations: (1) data movement is already a problem in DL; (2) this problem is getting worse with time as the technology trends are biased towards computation; and (3) not all data movement is the same—the farther the data movement the heavier the

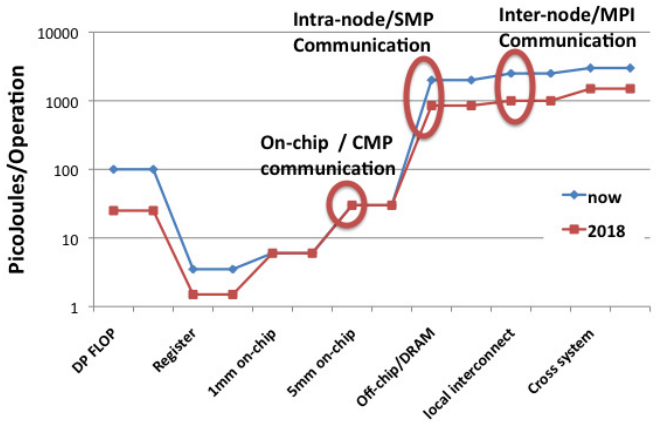

Figure 1.2: Energy used in data movement [143]

cost, both with respect to time and energy.

1.2.2 Data Processing Challenges

Together with data movement, data processing is an equally challenging facet of DL computation. The parallel DL ecosystem comprises multiple data-processing components, which typically run simultaneously and asynchronously on different computing devices in order to allow for high resource utilization and high computation throughput. There are two key challenges in data processing. First, we have to ensure that each individual data-processing component works at its full capability. Second, we have to make sure that all components work collaboratively and efficiently as a whole. Poor coordination between these components can cause *high hardware resource contention*, which leads to ***computational imbalance*** and prevents DNN training from achieving high scalability on large-scale systems.

Because of the graph-based computational model used in modern DL frameworks, such computational imbalance occurs only in portions of the graph where the resource demand is higher than the available hardware resources. This computational imbalance then propagates the resource contention into future iterations of the computation, thus further slowing down

the overall computation. Due to the subtlety of this problem, the DL community continues to overlook this scalability limitation in DL frameworks.

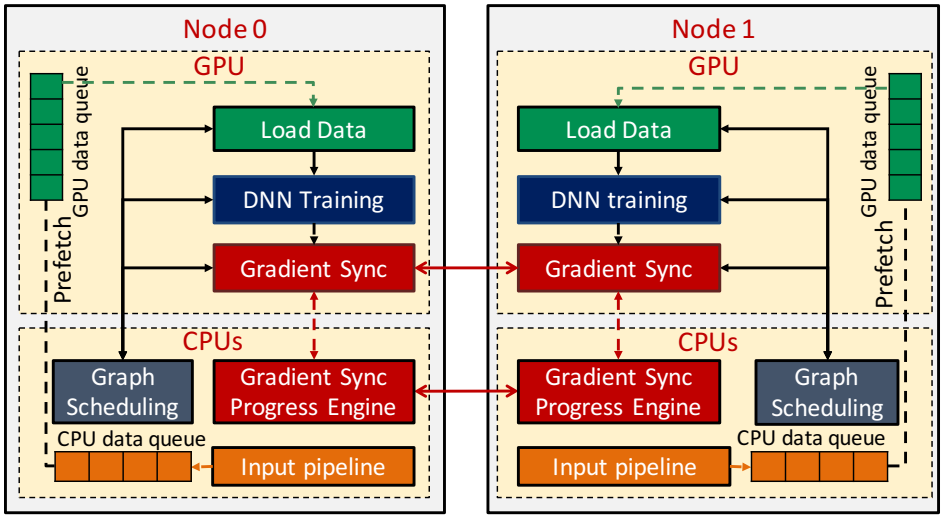

Figure 1.3: Data processing components in a multi-GPU multi-node DL system

At the present time, DL is typically executed on multiple GPUs across multiple nodes via a multilevel parallel DL model (i.e., with data parallelism across GPUs and model parallelism within each GPU). Figure 1.3 shows the data-processing components of the data-parallel environment on a distributed GPU cluster. In the figure, we show only one GPU per node for simplicity, although multiple GPUs can be used on each node in the real system. The modern parallel DL model on the GPU cluster mainly comprises four data-processing components: graph scheduling (on the CPUs), neural network training (on the GPUs), gradient synchronization (on both the CPUs and GPUs), and input pipeline processing (on the CPUs).

Here, components that share the same resource can potentially contend with each other. For instance, the gradient synchronization progress engine, the graph scheduling, and the input pipeline can compete with each other for CPU resources. The gradient synchronization (the Allreduce operation) can contend with the host-to-device input batch transfer (which is a part of the input pipeline) for the direct memory access (DMA) engine. The Allreduce

operation can compete with the neural network training for GPU resources.

Together with the direct contention within each hardware resource, the effects of such contention can also be *transferred* across hardware devices. For example, any delay in graph scheduling (on the CPUs) can slow down the neural network training (on the GPUs). The gradient synchronization progress engine's delay (on the CPUs) can affect the underlying tensor transfer (on the GPUs).

It is important to note here that the GPUs are the main computing units in the distributed GPU cluster. Thus, any slowdown in the GPU computation can significantly affect the overall execution performance. In parallel DL, GPUs are periodically synchronized with each other (e.g., through an Allreduce operation). At the synchronization point, if some GPUs are delayed due to the *direct* and/or *transferred computational imbalance* mentioned above, the others GPUs in the system have to wait, i.e., be idle. This can cause significant performance degradation in terms of scalability and computation throughput.

1.3 Thesis Big Picture

Based on the challenges presented in the previous sections, this thesis draws on sample problems from each challenge and studies them in detail: (1) data movement from the filesystem (i.e., file I/O)—presented in Chapters 3, 4, and 5, and (2) computational imbalance in data processing—presented in Chapter 6.

1.3.1 Investigation of File I/O

Most file I/O subsystems for DL at the present are similar to one another. In other words, they mostly suffer from the same shortcomings, for example, the use of `mmap` for partial

database access, the sequential database access limitation, and the lack of the true parallel data reading. We further compare and contrast the existing file I/O subsystems for DL to point out that our work can also be generalized for other systems in Chapter 7.2.1.

In our investigation of file I/O challenges in large-scale DL, we adopt Caffe, the most well-known first generation DL framework, with Lightning Memory-Mapped Database (LMDB) [27], the most widely used file I/O subsystem in DL frameworks to demonstrate the parallel data reading challenges. We perform an in-depth analysis on Caffe with LMDB to establish a clear understanding of the I/O problems in DL. Based on our analysis, we present many shortcomings in LMDB that are caused mainly by its usage of implicit I/O through `mmap` [103], its reliance on a tree-based structure for storing data that limits the database access to only sequentially, and its inefficiency in I/O management in the context of parallel computing. Some of these problems exist as the fundamental problems in other well-known file I/O frameworks, e.g., NumPy, TFRecord, RocksDB, and HDF5, as well. Our analysis shows that LMDB achieves less than 10% of the practically achievable performance of the I/O subsystem because of these shortcomings. Next, we propose "LMDBIO," an optimized I/O subsystem for scalable DL. LMDBIO includes a series of six sophisticated optimizations that address the shortcomings identified in our analysis.

According to the areas of the problems found in the analysis, we divide our optimization techniques into three main categories as follows:

- **Intra-node file I/O optimization** reduces high interprocess contention caused by `mmap` on a single node.
- **Speculative distributed file I/O optimization** coordinates between multiple reader processes to overcome the sequential database access limitation.
- **Direct file I/O optimization** is a collection of four techniques that adopt direct I/O

for data reading instead of the implicit I/O to improve data reading performance.

We showcase our file I/O optimizations on two large HPC clusters, namely Blues[1] and Bebop[2] at Argonne National Laboratory. Together, LMDBIO optimizations can significantly improve the performance of parallel data reading resulting in up to 65-fold improvement on DNN training on 9,216 CPU cores. In fact, on our system, these optimizations can saturate the system's available I/O bandwidth in DL frameworks.

We note here that the central focus of this study is on scaling DL on large supercomputing systems rather than on commodity clusters or cloud platforms. Most large supercomputers do not have a local disk on each node; thus, I/O typically is performed over the shared filesystem. In some cases, on-node storage might be present in the form of nonvolatile storage. Such storage is not persistent across the lifetime of the machine, however, and typically is wiped clean when a new job is assigned to a node. Thus, data I/O still has to be performed from the global filesystem. Even on systems that utilize on-node storage technologies in the form of burst buffers, staging data on to these burst buffers requires prior knowledge as to which node would need what segment of the data. Such information is, unfortunately, not readily available in modern DL systems.

1.3.2 Investigation of Computational Imbalance in Data Processing

The parallel data-processing architecture of modern DL frameworks are similar as presented in Chapter 1.2.2. Our study is applicable to all DL frameworks that allow asynchronous execution (or pipelining) of multiple data-processing components, for example, TensorFlow,

[1] http://www.lcrc.anl.gov/systems/resources/blues
[2] http://www.lcrc.anl.gov/systems/resources/bebop

PyTorch, and MXNet. We further discuss in Chapter 8.3.5 on how to apply our work in the other modern DL frameworks.

In our investigation of the computational imbalance challenges in large-scale DL, we showcase our study via TensorFlow, the most popular DL framework today, and Horovod [142], a well-known communication plugin for DL frameworks. We analyze the root cause of the scalability limitation and identify that it is not caused by the native performance of the hardware or software ecosystem itself but, instead, is an artifact of subtle resource contention issues that lead to *straggler processes* or *imbalance* in the amount of time spent on computing by the different processes. The parallel DL ecosystem comprises multiple data-processing components, which typically run simultaneously and asynchronously. Without proper coordination, these components can compete for resources leading to computational imbalance and limiting the DL training scalability.

Based on our analysis, we realize that Horovod, which is a part of the gradient synchronization component, is the main origin of computational imbalance. Therefore, we propose, design, and implement four optimizations that are incorporated into Horovod to allow multiple data-processing components to share the available computational resources more effectively in TensorFlow with Horovod. The following are brief descriptions of our proposed solutions:

- **Global sleep time optimization** solves the computational imbalance problem that is caused by nonuniform sleep time across processes by using a global sleep time.
- **Nonblocking cache synchronization** leverages a nonblocking approach to perform synchronization between processes during Horovod's cache synchronization to avoid resource contention (details of cache synchronization will be explained in Chapter 6.2.2).
- **Static CPU resource partitioning** avoids resource contention between data-processing components by using a simple static partitioning of the CPU cores.

- **Graph topology exploitation** makes use of the available graph topology information in order to prevent Horovod from competing for resources with other data-processing components.

We demonstrate the performance improvement achieved by our optimizations on the world's fastest supercomputer—Summit[3] [66] at Oak Ridge National Laboratory. Our optimizations can achieve up to 35% improvement in training various real-world DNNs using 24,576 GPUs of Summit.

1.4 Organization of this Thesis

The rest of the thesis is organized as follows. Chapter 2 presents background knowledge that is necessary to understand our work. Chapter 3 presents a detailed analysis of inter-process contention in LMDB on a single node and the design and implementation of our intra-node file I/O optimization. Chapter 4 elaborates on the analysis of LMDB's sequential database access restriction and the design and implementation of our inter-node file I/O optimization. Chapter 5 describes implicit I/O inefficiencies in LMDB and the design and implementation of four direct I/O techniques. Chapter 6 describes the resource contention problem that is caused by Horovod and the design and implementation of our solutions to the computational imbalance issue for DL data processing. Chapter 7 presents other literature related to our work and compares and contrasts our work with them. Chapter 8 summarizes the work done in this thesis and presents discussions on how to further improve the scalability of large-scale DL systems in several aspects.

[3]Ranked one on the Top500 list as of June 2019 (https://www.top500.org/lists/2019/06/)

Chapter 2

Background

This chapter provides brief background information that is necessary for understanding our analysis and the core design of our work.

2.1 Overview of Deep Neural Network Training

DL is one form of machine learning that uses high-order methods to recognize complex data and trends similar to the human brain. Machine learning is one approach to artificial intelligence (AI) which investigates ways for computers to learn from a set of data. Machine learning consists of two phases (as shown in Figure 2.1): (1) **training phase**: automatic complex pattern recognition based on given empirical data, and (2) **testing phase**: output prediction based on a given input using the model obtained from the training phase.

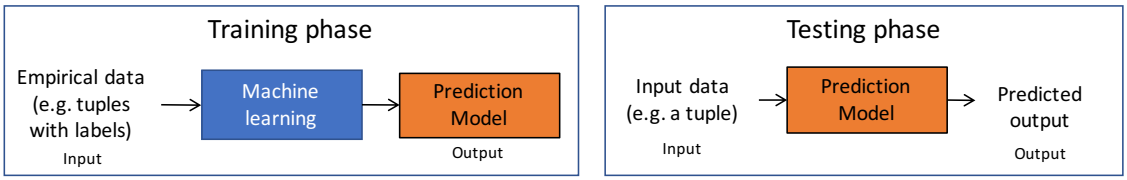

Figure 2.1: Training and testing phases of machine learning

DL can be viewed as a complex curve fitting problem. Although below we explain DL through classification examples, DL applications are not limited to only classification problems. One such simple curve fitting example is a binary classification where the input can be categorized

into two classes. Figure 2.2 shows the binary classification example that input data is a set of x and y pairs where x is the feature and y is the output which has two values, 0 and 1 in this case. These two values of y represent the two classes that data can be classified into. A simple way to classify the input is to use a linear line, $y = (ax + b >= 0)$. The goal is to find the optimal equation coefficients a and b that would allow the linear equation to classify the given input into two classes correctly. In the training phase, a simple method, such as linear regression, can be used to compute the optimal values for a and b. In the testing phase, the linear equation obtained in the training phase is a classification model that can be used to predict the class of an unseen input by simply plugging in the x value of the input into the linear equation.

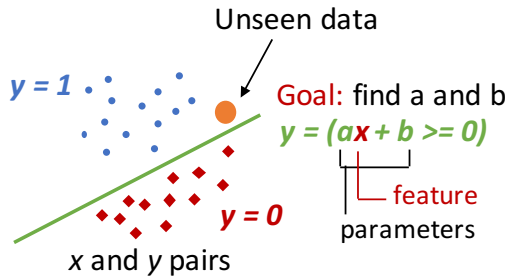

Figure 2.2: Binary classification example

For complex input data, a simple model typically fails to correctly classify data. Therefore, a more complex classification model is required. Figure 2.3 shows complex classification examples where high-order polynomial models are used to represent the classes of data. These models can be derived by using complex classification approaches [60], such as k-Nearest Neighbors (k-NN), Support Vector Machine (SVM), decision tree, random forest, as well as neural network.

Neural network (NN) is one of the well-known models for handling high-dimensional data. A neural network is made up of nodes and edges. Each node is called a "unit". Units are contained within layers. There are three types of layers in a NN (as shown in Figure 2.4):

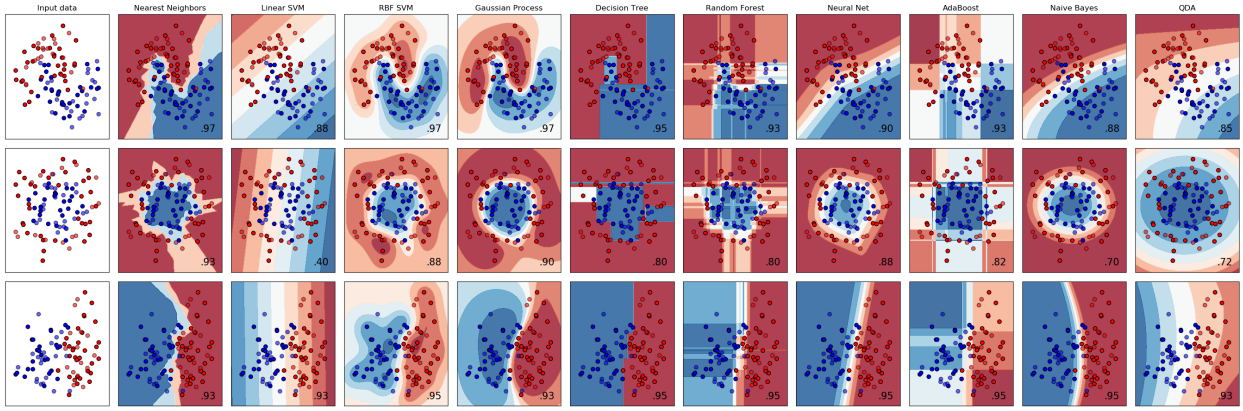

Figure 2.3: Complex classification example[a]: classification decision boundaries are shown in different colors

[a]Courtesy of https://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html

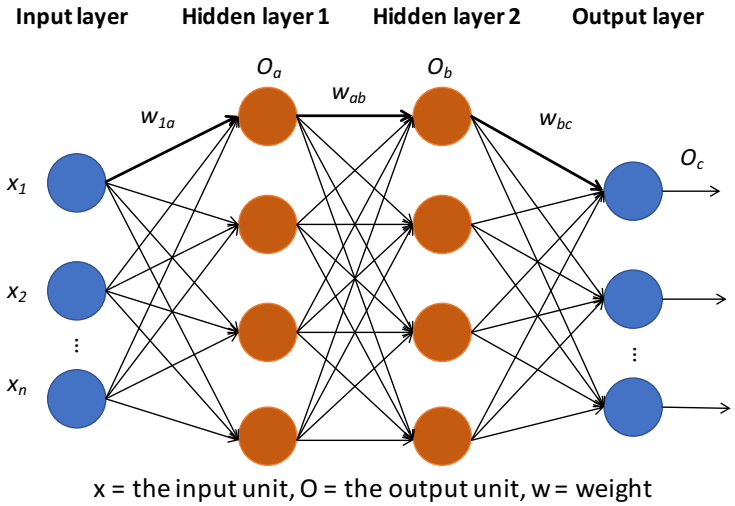

Figure 2.4: Structure of a neural network

1. **Input layer:** There is only one input layer in each NN. It is the first layer of the NN. All units within the input layer pass inputs simultaneously to units in the second layer.

2. **Hidden layers:** Each NN comprises one or more hidden layers. Units in this layer, called neurons, apply certain functions, called activation functions, to the weighted input (this will be explained later in this section). The last hidden layer passes the output to the output layer.

3. **Output layer:** There is only one output layer in each NN. It is the last layer of the NN. Units in this layer are called neurodes. This layer does the same thing as the hidden layer. Its product is the output of the NN.

On each edge, there is a weight that is a parameter (similar to a coefficient in a linear equation described above) of the NN model. It is used for adjusting the importance of the input value to the output by multiplying the weight value to the input value (e.g., if a weight is zero, the input is not important and will not be used in the calculation of the output value). For this reason, the configuration of the weight values is very crucial to the NN's output prediction ability.

To train a NN, a learning algorithm, such as stochastic gradient descent (which will be explained in the next section), iteratively adjusts parameters by minimizing an error between the predicted output and the actual output based on the input training data. The ultimate goal of the NN training is to obtain the optimal set of parameters that allows the NN model to achieve high *testing* (or *inference*) accuracy—which means that the NN model has high confidence in predicting the output of an *unseen* input data correctly. While high *training* accuracy means the NN model is highly capable to accurately estimate the output of the *trained* input data, it does not guarantee that the inference accuracy is also high. In the case that the training accuracy is high but the inference accuracy is low, it is an indicator

of an overfitting problem (i.e., the model is too specific to the training data).

Deep neural networks are modern neural networks that consist of larger number of hidden layers, i.e., deeper, and more recent activation functions compared to the traditional neural networks. Although the ultimate goal of the DNN training is as same as the traditional NN training, the complexity of the DNN training is much higher due to the more complex computation resource demand that is caused by an increase in the volume of training data, the more complex neural network structure and the more sophisticated training algorithms.

2.2 Stochastic Gradient Descent via Batch Training

In this section, we provide the overview of the stochastic gradient descent [52] (SGD) algorithm via the batch training approach. We omit mathematical explanations related to SGD as they are not used in the core of this work, however, they can be found in [115].

SGD is the most well-known algorithm used in DNN training. It is a backpropagation based method, that is it *iteratively* and recursively updates weights of the DNN model based on the prediction error. The traditional SGD algorithm involves a single data input in each training iteration, i.e., SGD learns only one data point before updating the DNN weights. Such training is oftentimes called "online training". However, online training is noisy as the weights are updated very frequently which can slow down the convergence [189]. Therefore, in common practice, more than one data samples—or a "batch" of data sample—is used in each training iteration. We henceforth refer to "batch SGD" as "SGD".

Figure 2.5 shows the SGD training workflow. The SGD training starts by initializing the parameters of the DNN. Most training frameworks typically initialize the parameters to random values, although a growing number of researchers use better initial approximations

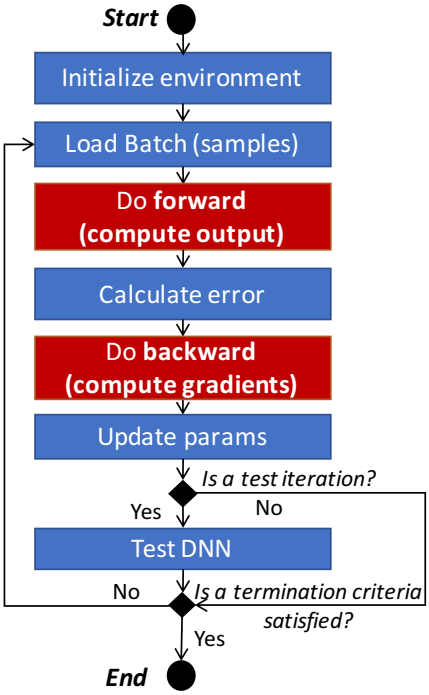

Figure 2.5: Stochastic gradient descent

of the parameters based on known properties of the input data. Training is an iterative process that continues until the parameter set converges to the desired accuracy. Processing one batch of data samples is referred to as one training iteration. In each training iteration, a batch of data samples in the database is drawn randomly and used to train the neural network in the *forward pass* computation. The forward computation calculates the output of each layer starting from the first to the last layer. Then the deviation error is measured between the predicted value from the current DNN parameters (i.e., the output from the forward pass computation) and the actual value from the dataset. This error is then propagated back into the DNN in the *backward pass* and used to calculate parameter adjustments or gradients. The backpropagation starts from the last layer to the first one. For the batch training, the gradients of all data batch samples are accumulated and used to update the weights. Once the training converges, the final set of DNN parameters is used to generate

a function approximation model that can be utilized for highly accurate prediction of new data samples.

2.3 Parallel Batch Training

As the computations of data samples in each batch is independent from one another, a parallel batch training is feasible. Most modern DL frameworks allow for parallel DNN training. There are four main parallel models (as shown in Figure 2.6):

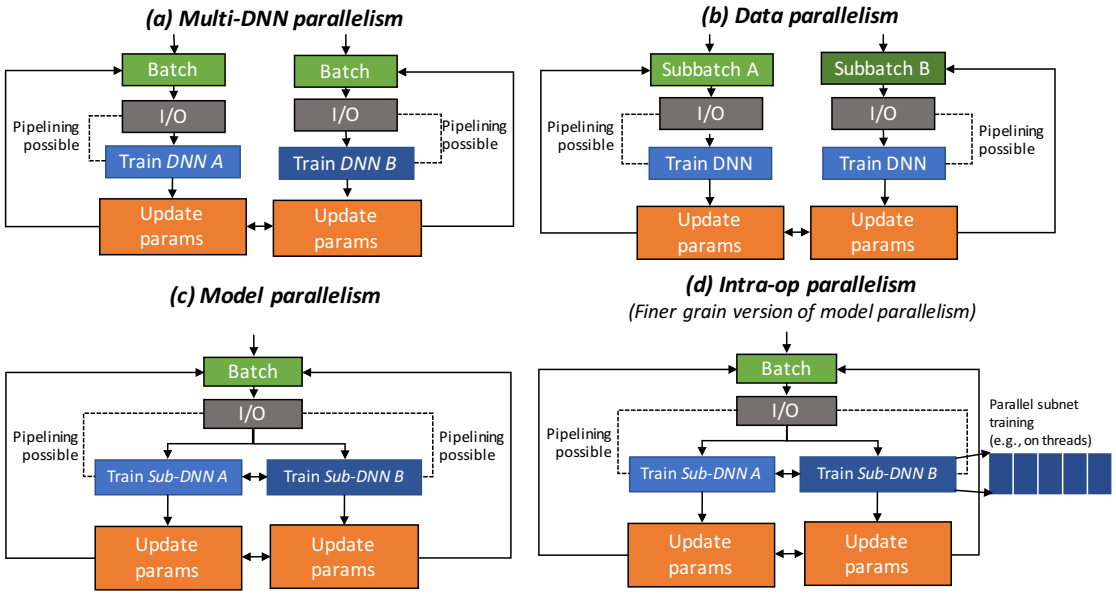

Figure 2.6: Parallel DL models: (a) multi-DNN parallelism; (b) data parallelism; (c) model parallelism; (d) intra-op parallelism

1. **Multi-DNN parallelism:** The different DNN models are trained concurrently across different processes/threads. The same batch of data is used for all the models. This kind of training is also referred to as "ensemble training" [92].

2. **Data parallelism:** This model is the most popular parallel model for DL. The batch of data samples is partitioned into a smaller chunks of the same or similar sizes, and

the different chunks are trained on multiple processes/threads simultaneously using the same copy of the DNN. In the parameter update step, gradients are accumulated across processes/threads (normally via the Allreduce communication operation) before updating parameters.

3. **Model parallelism:** This model is suitable for the cases where the DNN model is too large to fit in the memory (e.g., main memory or GPU's memory) or asynchrony in data parallelism is not sufficient. In this parallel model, the DNN model is partitioned and executed in parallel. The same batch data is used for training the different replicas of the DNN. Communication between the DNN replicas occurs based on how the DNN is partitioned.

4. **Intra-op parallelism:** This model is a fine-grained version of model parallelism. Within each DNN, each operation is partitioned and run concurrently on multiple threads/processes.

For every parallel model, the input component (labeled "I/O" in Figure 2.6) can be pipelined with the DNN training, for example, the input component is running on the CPUs while the DNN training is running on the GPUs. These parallel models can be combined together to maximize parallelism in the DNN training. Moreover, most modern DL frameworks support all four parallel models.

2.4 Overview of Modern Deep Learning Software

We provide the overview of the DL software adopted in our work. In our file I/O optimization, we use the Caffe DL framework with its default file I/O subsystem, LMDB. While

in our computational imbalance optimization, we use TensorFlow and Horovod as our DL framework and network I/O subsystem, respectively.

2.4.1 Overview of Caffe Deep Learning Framework

Caffe [75] is a one of the very first DL frameworks developed by the Berkeley Vision and Learning Center. Caffe is a C++ based DL framework with CUDA support. The original goal of Caffe was to provide an efficient GPU-based framework for convolutional neural network training [52], but it has been extensively modified by several researchers to support generic CPU architectures as well.

Caffe adopts the data parallel SGD approach to train DNNs as shown in Figure 2.7. The overall flow of the training is the same as that of sequential processing except that the data batch loading, the forward pass, and the backward pass are parallelized on multiple processes/threads. Parallel neural network training, however, comes with an additional communication cost where neural network parameters must be synchronized across processes/threads. For storing and retrieving data samples, a number of database options are available in the community for DL systems. The most widely used database option is LMDB, which is the default database format used by Caffe. As Caffe is the first generation modern DL framework, it does not support asynchronous execution of the input and the gradient synchronization components. All data-processing components are executed sequentially as illustrated in Figure 2.7. In other words, the I/O-computation overlap and the communication-computation overlap are not provided in Caffe.

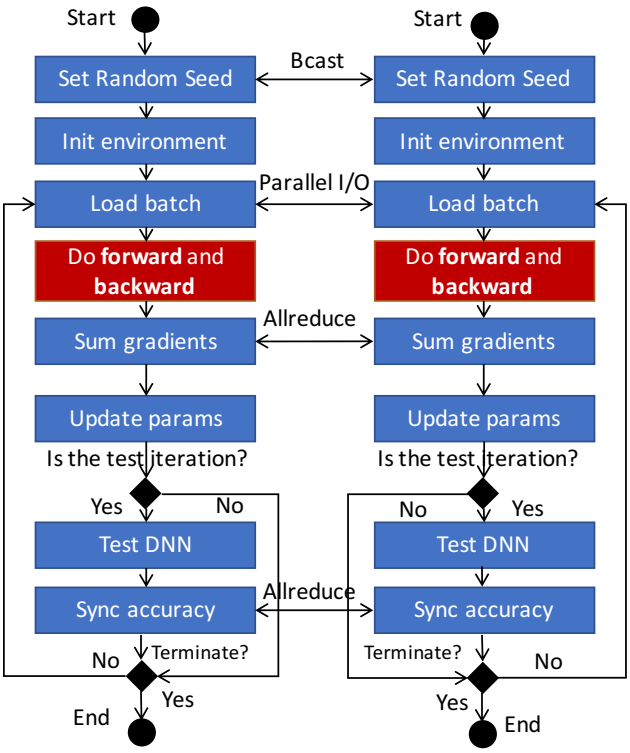

Figure 2.7: Caffe's data-parallel workflow

2.4.2 Overview of Lightning Memory-Mapped Database (LMDB)

LMDB involves two concepts. First, it refers to a database format that arranges its content based on a B+ tree and allows efficient simultaneous read and write access to the database. Second, LMDB refers to a library that provides the API to access and manipulate the LMDB database. This library makes use of the OS memory-mapping mechanism, `mmap`, to enable in-memory database access. We note that LMDB is not specific to DL. It is a well-known key-value database that is used in multiple domains with different usage models. In this section, we discuss background information related to LMDB in two ways: (1) `mmap` and its dynamic data-reading mechanism and (2) the LMDB database format and its data access model.

Overview of LMDB's Dynamic Data Loading via `Mmap`

LMDB relies on `mmap` to perform in-memory data access. `Mmap` is a generic Unix system call that maps the layout of a file on the filesystem to the virtual address space of a process, thus giving an illusion to the process that the entire file is in memory. Data access is tracked by the OS at a page-level granularity, and data is dynamically fetched from the filesystem to memory when the application accesses it. This model is convenient for accessing files with complex structures, such as B+-tree databases since the application process does not have to be concerned about which exact bytes need to be fetched to memory. It can pretend that the entire file is already in memory.

`Mmap` dynamically reads pages from the filesystem to memory on demand. In other words, the data is not read until it is required. The workflow used by `mmap` is illustrated in Figure 2.8. Three components are involved in data reading: the filesystem (which can be local or shared across machines), a page cache (which is shared across processes on the node), and a virtual

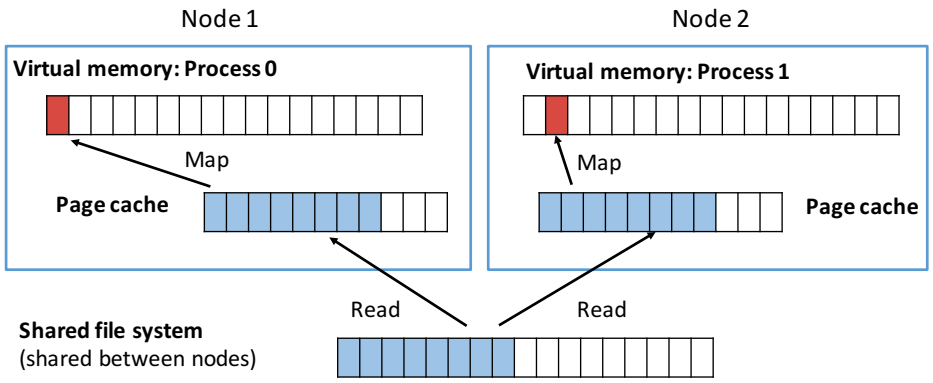

Figure 2.8: Workflow of memory-mapped file I/O (`mmap`)

address space (which is private to each process). When `mmap` is called, each process allocates a virtual address space for the database, but it does not fetch any data into this space. Instead, the `mmap` call protects the allocated virtual address space to raise a page fault if the process tries to read from or write to this virtual address space. When the process accesses the first page, which is not present in memory, the page fault handler is triggered. The page fault handler first reads data from the filesystem to the page cache and then maps the corresponding page from the page cache to the appropriate virtual address region that the user is trying to access. The pages in the page cache can be mapped to the virtual address spaces of multiple processes on the same node. Another benefit of using `mmap` is that the OS automatically frees pages when physical memory is almost full. Thus, the user does not have to worry about out-of-memory problems.

Aside from these advantages, `mmap` also has several shortcomings that stem primarily from the fact that it offloads all the I/O handling responsibilities to the OS. Thus, application processes do not have any control over the actual I/O. For example, `mmap` does not allow users to provide detailed information about their access pattern. While users can provide some simple hints using `madvise` and `fadvise`, these hints are primarily for simple manipulation of access patterns. For example, they allow users to distinguish between sequential and

random access to data. However, they are not suitable for more complex access patterns, such as strided access to batches of data. This abstraction of I/O from the applications that `mmap` provides sometimes results in a tremendous loss in I/O performance.

LMDB Database Format

LMDB adopts a flattened B+-tree data structure as its database layout. It uses pages to represent nodes (i.e., branch and leaf nodes) in the B+ tree, where each node is stored on the filesystem in a block-aware manner.

B+ trees are balanced *n-way* search trees. LMDB uses this tree format to organize the database indices in a way that data records can be accessed efficiently when stored on a local or external filesystem. Generally speaking, a B+ tree consists of two types of nodes: *branch nodes* and *leaf nodes* (see Figure 2.9 for a 3-way B+ tree structure). A branch node contains pointers that point to n children nodes (which can be branch nodes or leaf nodes). Indices contained in a branch node govern the range of indices of its successors. For example, in Figure 2.9, the pointer from index 3 in the branch node points to a leaf node that contains data with indices less than or equal to 3. B+ trees are designed to be efficient for filesystem access. In B+ trees, nodes are stored in a block-aware manner, where each node is a filesystem page.

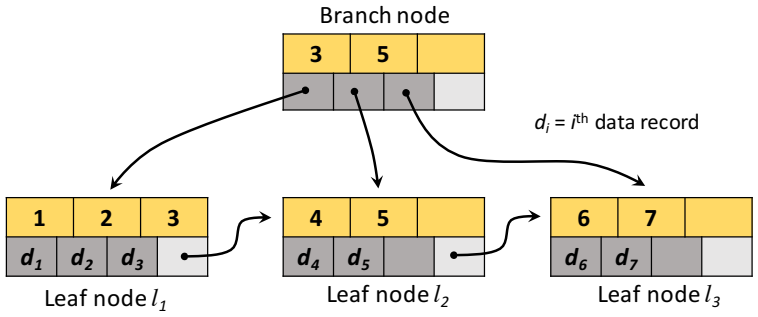

Figure 2.9: B+ tree data structure

The LMDB database consists of four types of pages: *metadata pages*, *branch pages*, *leaf pages*, and *overflow pages*. The first two pages of the database file are metadata pages that store information specific to the overall database (e.g., version of the database, size of the database). The branch and leaf pages represent the internal branch and leaf nodes in the B+-tree structure. Each of the branch and leaf pages contains a page header that has information associated with that page (e.g., type of page, amount of free space in the page, pointers to neighboring pages) and some actual data key-value records. Since the size of each page is typically limited to 4 KB (the OS page size), a leaf node cannot accommodate data records that are larger than 4 KB.[1] In such cases, LMDB uses overflow pages to store data records that cannot fit within one page. Thus, each leaf page can have zero or more overflow pages associated with it. We note that not all overflow pages have a header associated with them. Only the first overflow page corresponding to a given leaf page has a header.

Since LMDB's data format is a complex tree structure, correctly identifying a record requires a complete collection of pointers to all the branch pages in the path to the target data record. LMDB stores this information in a convenient data structure that it refers to as the "cursor," which can be thought of as the identity of a data record in the LMDB database. When the database is opened, LMDB initializes this cursor to point to the root of the database. LMDB also provides API functions to move the cursor forward or backward, thus allowing us to access the remaining records in the database.

LMDB's database format is designed to allow for efficient sequential data access: each leaf node has a link that connects it to the adjacent leaf node. The layout of the LMDB database file depends not only on database's contents but also on how the content was inserted into the database, that is, the order in which they are inserted and the frequency of database commit operations between insertions. In other words, for a given set of data records, depending on

[1] Even though most systems today support large 2 MB pages and huge 1 GB pages, file-backed `mmap` still typically uses 4 KB pages, even on modern systems.

how they are inserted and how frequently the commit operations are issued, the database layout can be very different. Thus, we cannot determine the exact layout of the database unless we also have information on how the database was created. This information is not stored in the native LMDB database format.

In order to access an arbitrary data record in the database, LMDB needs to navigate from the root of the B+ tree and through all the corresponding branch and leaf nodes, but not the overflow nodes. Such tree parsing, which we refer to as "sequential seek," requires data to be read from the filesystem to memory because the pointer information is stored in the page headers. Although only the header portion of the page is needed for parsing the tree, the entire page is read to memory since `mmap` loads data at a page-level granularity. The worst-case scenario is when every data record fits in the leaf page (i.e., no overflow pages). In this case, every page along the way to the target leaf page will need to be read while parsing the tree.

Since not all pages have headers associated with them (e.g., overflow pages), none of the existing LMDB operations allow for random access within the database. Unfortunately, the data access pattern of parallel DL is semi-random—in other words, each process would need to skip the records that are being processed by the other processes—thus making file I/O hard to optimize in such frameworks.

2.4.3 Overview of TensorFlow Deep Learning Framework

TensorFlow [6, 166] is the most popular open-source machine learning framework to date. It is developed by Google. It provides various highly optimized machine learning building blocks that can be easily utilized via Python and C++ APIs. TensorFlow can operate on several computing platforms ranging from mobile devices to supercomputers. It supports all

four parallel DL training mentioned in Chapter 2.3. The details of the parallel model used in this work will be explained in Chapter 6. Unlike Caffe, TensorFlow provides asynchronous execution of data processing-components. In common practice, the input component is pipelined with the core DNN execution when possible. Moreover, the communication and computation overlap can also be enabled in TensorFlow, that is the gradient synchronization process can be pipelined with the backward pass computation.

2.4.4 Overview of Horovod Communication Plugin

Horovod is a communication plugin for distributed DL frameworks, including TensorFlow, PyTorch, and MXNet. It provides new communication operation classes with similar semantics as the native communication operations in these DL frameworks. Because of this, users can construct *computation graphs*, which are the representations of the DNNs, by simply replacing the native communication operations with Horovod operations.

Horovod relies on several highly optimized data-movement libraries, such as MPI, NVIDIA Collective Communications Library (NCCL),[2], IBM PowerAI Distributed Deep Learning (DDL) [26], Facebook Gloo,[3] and Intel Machine Learning Scaling Library (MLSL) [149] for communication, thus allowing for better data-transfer performance and scalability. Each Horovod operation takes one input tensor and produces one output tensor. Typical DL computations require processing more than one tensor. Consequently, computation graphs generally contain multiple Horovod operations.

All Horovod operations are *nonblocking asynchronous*. They are *nonblocking* in that a Horovod operation will always return in a finite amount of time, irrespective of the state of other processes in the system. Specifically, when a Horovod operation is executed in the

[2]https://developer.nvidia.com/nccl
[3]https://github.com/facebookincubator/gloo

computation graph, the DL framework's graph scheduler thread enqueues a communication request into the Horovod's internal request queue and then returns. Horovod operations are *asynchronous* in that once the operation is enqueued, the DL framework is no longer responsible for its completion. The progress and completion of the operation are asynchronously handled by Horovod. To achieve this, within each OS process, Horovod creates a "background thread" whose primary purpose is to perform the data transfers associated with the various Horovod operations in that process. The background thread periodically checks the request queue (into which the graph scheduler had enqueued communication requests), issues data transfers for the tensors associated with the enqueued requests, and executes completion callbacks to the DL framework once the transfers are completed. More details of the background thread will be explained in Chapter 6.2.2.

Chapter 3

Intra-node File I/O Optimization

In this chapter, we first provide the scalability analysis of Caffe with its default I/O subsystem, LMDB, to demonstrate data reading inefficiency in the state-of-the-art DL framework. We then provide the detailed analysis of LMDB inefficiency on a single node along with the detailed design and implementation of our proposed optimization, namely **LMDBIO-LMM**—localized *mmap* optimization.

From the analysis, we observe that data reading performance of LMDB is inferior when using multiple readers within one node. In fact, using multiple readers is worse than using a single reader. This is because LMDB internally uses `mmap` which entirely relies on the OS to handle I/O operations. The low quality I/O management of the OS causes high interprocess contention when using multiple readers. This results in a surge of a number of context switches in the multiple-reader environment.

LMDBIO-LMM attempts to eliminate interprocess contention of `mmap` by localizing it. Our `mmap` localization approach reduces I/O management stress of the OS which results in a significant increase in overall reading performance within a single node.

At the end of this chapter, we compare the performance of LMDBIO against that of LMDB using a microbenchmark and actual DL benchmark on various datasets and supercomputing platforms. Our experimental results show that LMDBIO performs better than LMDB in most cases.

3.1 Analysis of Caffe/LMDB Performance and Inefficiencies

In this section, we analyze the scalability performance of Caffe with LMDB as well as the root cause of the scalability loss.

3.1.1 Experimental Setup for File I/O Experiments

To enable cross-comparison between proposed file I/O optimization techniques in our proposed work (including the approaches presented in Chapters 3, 4, and 5), we use the same experimental setup for all experiments. Our evaluation framework addresses several dimensions of large-scale DL including size of dataset, type of dataset, type of neural network, type of supercomputing platform, type of data storage, type of software stack, data access pattern, and configuration of the experiment. In this work, we fix the latter four dimensions while experimenting with the rest, as noted in the following subsections.

Datasets: We use three *image classification* datasets for our experiments. The first is the **CIFAR10-Large**[1] [83] dataset, which consists of 50 million sample images in 10 classes, each approximately 3 KB. The total dataset size, including raw images and metadata corresponding to the images, is approximately 190 GB. The second is the **ImageNet**[2] [138] dataset, which consists of 1.2 million sample images in 1,000 categories, each approximately 192 KB (total dataset is 240 GB). The third is the **ImageNet-Large** dataset, which is

[1]CIFAR10-Large is an amplified version of CIFAR10 which is a dataset that contains 60,000 32x32 color images in 10 classes. Although we adopt simple replication techniques to augment our dataset, other data augmentation approaches (such as Gaussian noise [133]) can be used to replicate our results as well because the size and the layout of the dataset would be identical irrespective of which technique is used. CIFAR10 can be downloaded from https://www.cs.toronto.edu/~kriz/cifar.html

[2]The ImageNet dataset that we used is a part of Large Scale Visual Recognition Challenge 2012 (ILSVRC2012). The dataset can be downloaded at http://www.image-net.org/challenges/LSVRC/2012/

an amplified version of ImageNet that replicates some images from ImageNet for a total of 6 million images (total dataset is 1.1 TB). Although all datasets can be I/O intensive, the ImageNet datasets are particularly so, given the size of the images that need to be processed.

DNNs: We use three DNNs for our experiments. **AlexNet**[3] [84] is used to train the CIFAR10-Large dataset. AlexNet is a small neural network with 13 layers and 89K parameters. **CaffeNet**[4] is used to train the ImageNet and the ImageNet-Large datasets. It is a large neural network with 22 layers and 60M parameters. **ResNet50**[5] [63] is used to train ImageNet-Large. It is a large network with 228 layers and 25.6M parameters [184]. We note that the number of layers in each DNN is based on the DNN definition in Caffe.

Supercomputing platforms: The experimental evaluation for this work is performed on two clusters operated by the Laboratory Computing Resource Center at Argonne National Laboratory: **Blues** and **Bebop**. Blues consists of 310 computing nodes connected via InfiniBand Qlogic QDR. Each node has 64 GB of memory, two Sandy Bridge 2.6 GHz Pentium Xeon processors (16 cores, hyperthreading disabled), and a 15 GB ramdisk. Bebop consists of 672 Intel Broadwell nodes. Each node consists of 36 cores, 128 GB of memory, and a 15 GB ramdisk. The interconnect is Intel OmniPath.

Data storage: The datasets are stored on IBM General Parallel File System (GPFS). Blues and Bebop share the same GPFS installation. The storage is 110 TB of clusterwide space. The system has 10 Network Shared Disk servers with no replication. To ensure that data is read from the filesystem rather than from memory (i.e., no caching), we clear the OS's page cache and GPFS's file cache prior to performing each experiment.

[3]The AlexNet neural network is Caffe's variant of AlexNet which is also known as "cuda-convnet". The neural network definition can be found at https://github.com/BVLC/caffe/blob/master/examples/cifar10/cifar10_full.prototxt.

[4]CaffeNet definition can be found at https://github.com/BVLC/caffe/blob/master/models/bvlc_reference_caffenet/train_val.prototxt

[5]https://github.com/KaimingHe/deep-residual-networks/blob/master/prototxt/ResNet-50-deploy.prototxt

DL frameworks and software stack: We use **Caffe** version 1.0.0-rc3 together with single-threaded Intel Math Kernel Library or MKL [167], unless specified otherwise. We use **LMDB** version 0.9.21.[6] Caffe and LMDB are built by using the Intel ICC compiler (version 17.0.4). On Blues, we use MVAPICH-2.2 over PSM (Performance Scaled Messaging) [160] for all experiments. On Bebop, we use Intel MPI version 2017 for all experiments.

Data access pattern: Caffe supports a data access pattern that is commonly known as data sharding, strided data reading, or distributed data reading. All processes together access contiguous blocks of data in each iteration, while each individual process has a strided access pattern for data access across iterations. Thus, each process accesses a disjoint set of data samples in each training iteration. This data access pattern is a part of the Caffe workflow and is the same across all experiments. Data reading and parsing are performed in parallel by multiple processes. We note that Caffe does not perform data shuffling.

Experimental configuration: All experiments use single-precision floating-point SGD training. All experiments are run three times, and the average performance is shown.

3.1.2 Scalability Analysis of Caffe/LMDB

In this section, we analyze the file I/O bottleneck in Caffe. We evaluate the strong-scaling scalability of Caffe with LMDB (denoted Caffe/LMDB) by training it using the CIFAR10-Large dataset on AlexNet [84]. We use a batch size of 18,432 for 512 iterations (approximately 9 million total data samples) on Bebop. The training is scaled from 1 core to 9,216 cores (i.e., 256 nodes with 36 cores on each node). Figure 3.1(a) shows the overall training time of Caffe compared with linear strong scaling. The figure shows that Caffe/LMDB scales poorly even with a small number of cores and is nearly 660-fold worse than linear strong scaling on

[6] https://github.com/LMDB/lmdb/releases/tag/LMDB_0.9.21

9,216 cores.

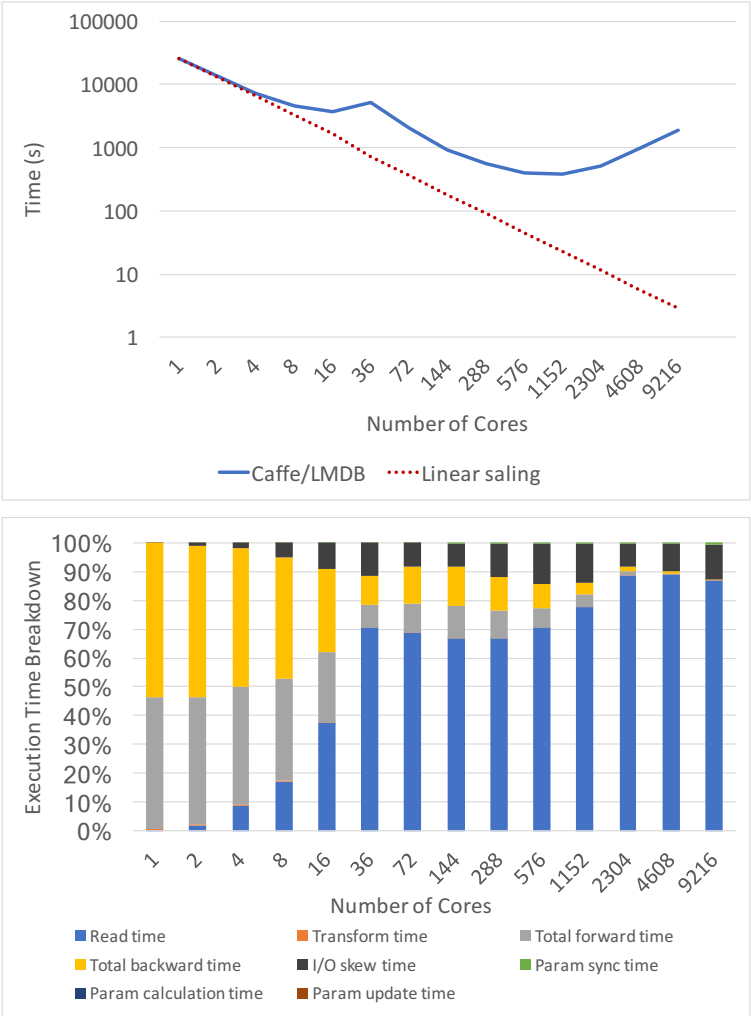

Figure 3.1: Caffe/LMDB's strong scaling using CIFAR10-Large on Bebop: (a) total execution time; (b) execution time breakdown

We next perform a breakdown of the execution time, shown in Figure 3.1(b), to understand which components of Caffe/LMDB take the most time. We notice two significant trends in the figure. First, the file I/O time (denoted "Read time") increases with the number of cores. This increase is because the computation time (i.e., total forward time, total backward time, parameter calculation time, and parameter update time) tends to scale well with the number of cores, leaving I/O as the bottleneck. In fact, as we scale the problem to 9,216 cores, I/O

takes approximately 90% of the total time. Second, the "I/O skew time" grows with the number of cores, raising concerns of load imbalance occurring between the cores as we scale to a large number of cores.

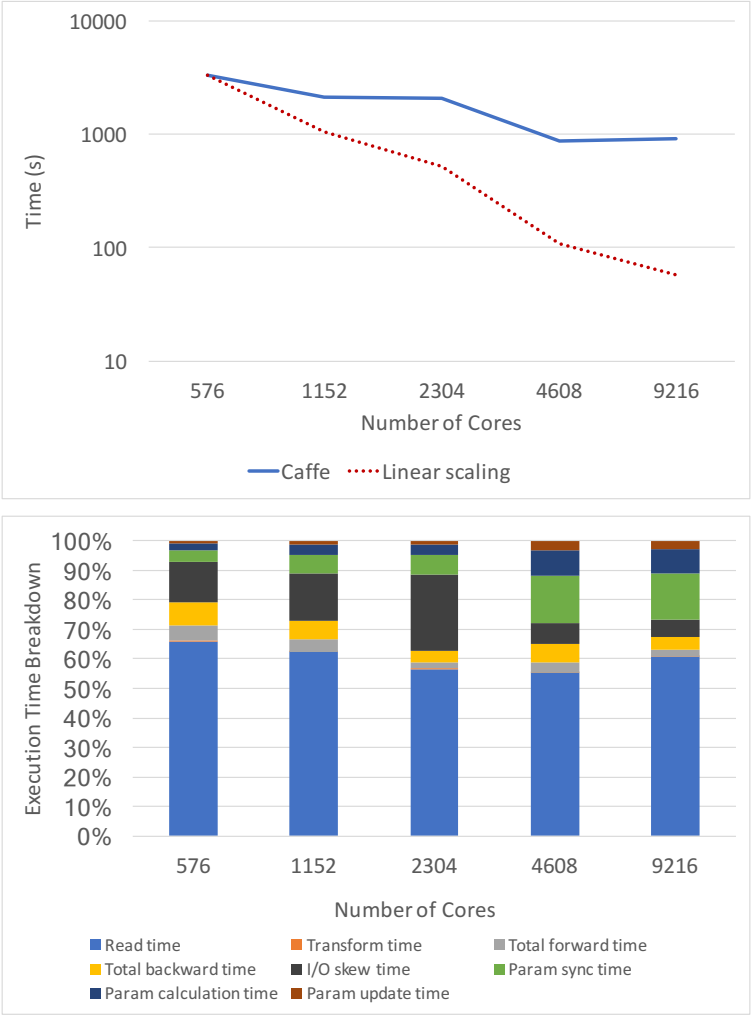

Figure 3.2: Caffe/LMDB's strong scaling using ImageNet-Large with CaffeNet on Bebop: (a) total execution time; (b) execution time breakdown

We also perform a similar analysis with ImageNet-Large using the CaffeNet and ResNet50 network models by using a batch size of 18,432 for 128 iterations, as illustrated in Figures 3.2 and 3.3. From our experiments, we observe that the actual read time (denoted "Read time") and the I/O imbalance time (denoted "I/O skew time") take up to 66% of the time with

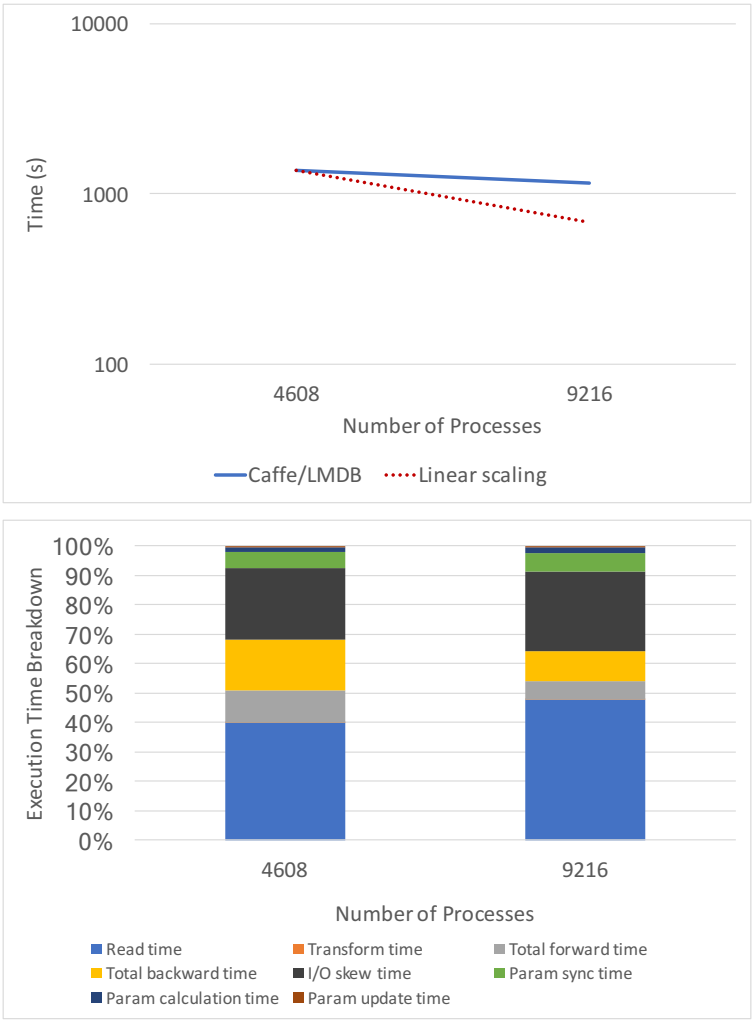

Figure 3.3: Caffe/LMDB's strong scaling using ImageNet-Large with ResNet50 on Bebop: (a) total execution time; (b) execution time breakdown

CaffeNet and 75% of the time with ResNet50 on 9,216 processes, thus dominating the overall training time. We note that we cannot run experiments with ResNet50 on less than 4,608 processes on Bebop because of insufficient memory on each node. For this reason, we use the CaffeNet neural network for all of the remaining experiments with the ImageNet-Large dataset in this work.

A broader issue that one needs to be aware of is that hardware technology trends point to the fact that I/O is already a bottleneck and its performance relative to that of computation is only getting worse with time [137]. The specific ratios between the computation cost (in the model that we choose) and the I/O cost (in the filesystem that we choose) are simply representative examples of a more general problem.

As a final step, we would like to understand the peak performance that our filesystem can achieve, to help us distinguish between using a slow filesystem vs. LMDB itself being inefficient. To do so, we use the IOR benchmark [144] to measure the performance of POSIX I/O on Bebop and compared that with the file I/O performance achieved by Caffe/LMDB. IOR performance is often considered to be the best case for I/O performance that a given platform can achieve. Our comparison shows that the I/O performance achieved by Caffe/LMDB is much worse than that reported by IOR. In fact, the file I/O bandwidth achieved by Caffe/LMDB is less than 10% of that demonstrated by IOR. This result suggests that the performance loss is caused mainly by inefficiencies in Caffe/LMDB rather than by limitations in the filesystem (or the I/O hardware) itself.

3.1.3 Memory-Mapped File I/O (`mmap`) Interprocess Contention

As discussed in Section 2.4.2, `mmap` maps the layout of a file from the filesystem to the virtual address space of a process and enables accesses to the file as if it were a memory buffer. `Mmap`'s fundamental workflow relies on the OS to dynamically fetch data from the filesystem to physical memory. The OS, however, has no knowledge that the application is a parallel application, and hence it must consider the `mmap` done by each process to be independent (except for sharing the page cache when possible). This behavior, however, causes an unfortunate interaction with the Linux process scheduler.

Before discussing `mmap` interprocess contention in detailed, it is noteworthy to understand Linux scheduler and its scheduling policy. Linux kernel 2.6 introduced the Completely Fair Scheduler (CFS) [114] as the default process scheduler. CFS guarantees fairness of CPU usage between processes and attempts to maximize CPU utilization. It schedules processes to execute on the CPUs from a red-black tree where a process with the least-used CPU time will be chosen to run first. The scheduler does not take into account the order in which the processes are enqueued.

Despite its various benefits, `mmap` suffers from a few shortcomings, specifically in the way it handles I/O requests. When a user process accesses a page, if that page is not already in memory, a page fault handler is triggered to fetch the data from the filesystem. This I/O request is then passed down to the hardware controller (e.g., Small Computer System Interface or SCSI [136] for local storage or a network I/O adapter for network-based filesystems), and the user process goes to sleep while waiting for the I/O request to complete. When the I/O operation completes, the hardware controller raises an interrupt informing the filesystem of the completion. We note here that this interrupt handler is a *bottom-half* handler in Linux. That is, the interrupt is not associated with any particular user process in the system but

is a generic event informing the filesystem that an I/O operation that was issued by one or more processes has now completed. The interrupt handler then marks as runnable all processes that were sleeping while waiting for an I/O event.

At this point, CFS takes over. The next time the scheduler is triggered, it traverses all processes in its red-black tree and schedules the runnable processes one at a time. In general, however, since only some of the processes were waiting for the specific I/O operation that just completed, most processes will see that their I/O operation has not completed and go back to sleep. Only one or a few processes will be able to use this I/O completion to perform further processing. Consequently, this model significantly increases the number of context switches that get triggered, with most of the switches resulting in no real work. This approach thus increases the amount of "sleep time" associated with each process as well.

This problem is not present when performing a simple `mmap` I/O with a single process accessing the data. In such cases, the I/O completion handler wakes up only one process, and every completion corresponds to the exact process that is waiting for that I/O operation. However, the larger the number of readers, the greater the chance that the processes will be woken up without having any real work to do. Therefore, we expect the total number of context switches in the processes to grow as we increase the number of readers. Moreover, we expect that with more than one process, the processes will spend most of the data reading time waiting for I/O (i.e., sleeping).

We demonstrate this behavior in Figures 3.4(a) and 3.4(b). As expected, the context switches increase as we increase the number of processes (note that there is one process per core). The number of context switches increases by approximately 48 times from one process to two processes and by approximately 18,000 times from one process to 9,216 processes. When using multiple processes to read the data, the ratio of the sleep time to the read time increases to 99% on 9,216 processes. These results show that `mmap`-based file access is highly inefficient

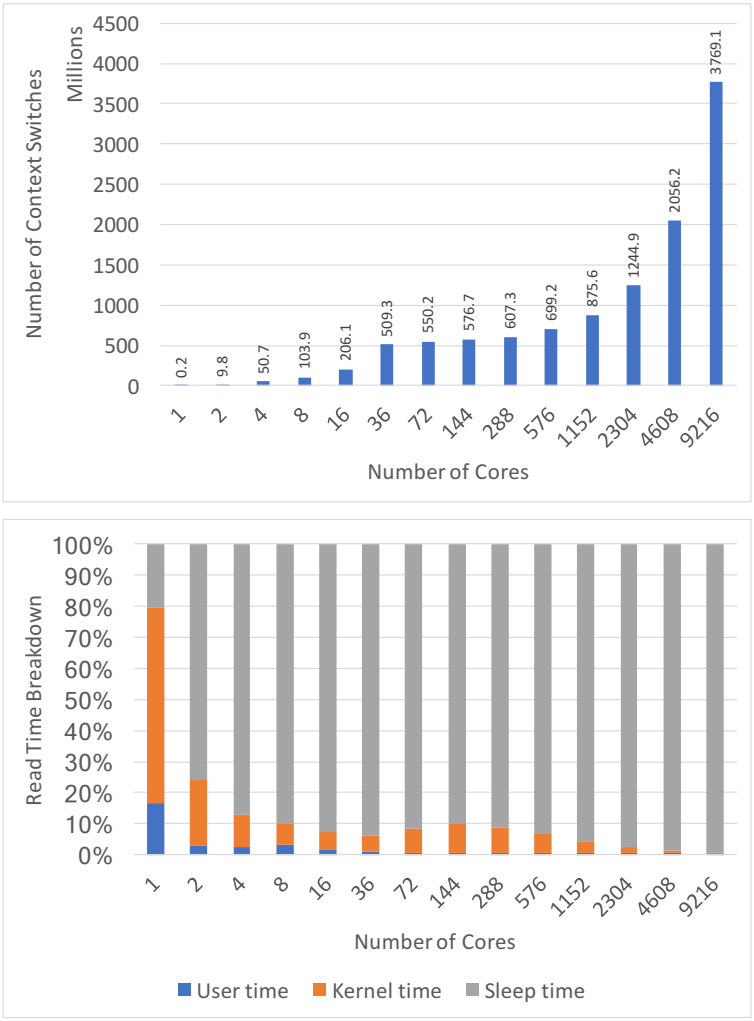

Figure 3.4: Caffe/LMDB's `mmap` analysis (CIFAR10-Large dataset on Bebop): (a) context switches; (b) sleep time

with more than one process because processes are wasting time in getting context switched in and out for I/O interrupts that do not belong to them.

3.2 Design and Implementation of LMDBIO-LMM: Localized `Mmap` Optimization

As discussed in Section 3.1.3, LMDB is highly inefficient in the way that it accesses data. The primary inefficiency comes from the way `mmap`-based I/O requests are generated and handled by multiple processes. To alleviate this issue, we propose LMDBIO-LMM—an I/O plugin for DL frameworks that takes the data access characteristics of Caffe into account in order to optimize the I/O performance.

One of the major drawbacks in the current file I/O model in Caffe is the lack of true parallelism. Data fetching for each process is independent of the other processes even though it is a parallel application and the I/O could, in principle, be coordinated. Thus several potential optimization opportunities are lost. Often, this approach results in unnecessary data to be read and discarded. Moreover, since all processes are performing file I/O and relying on the underlying bottom-half handler to wake them up, the wakeup model is imprecise and leads to unnecessary context switches.

The general idea of LMDBIO-LMM is to utilize what we refer to as "localized `mmap`." In this model, a single process is chosen on each node as the root process. The root process reads data from the filesystem and distributes it to the remaining processes on the node using MPI-3 shared memory. This approach aims to reduce I/O parallelism in order to give `mmap` a more sequential view of I/O and to minimize interprocess contention. The `mmap` localization approach also allows the traditional Linux bottom-half handler for I/O to wake up the exact process that is waiting for I/O to complete since only one process is performing I/O. This strategy minimizes the number of context switches and helps improve performance.

LMDBIO-LMM is written in C++ and utilizes MPI and LMDB as core engines. We assume

the availability of MPI-3[7] in order to allow LMDBIO-LMM to detect process colocation automatically, perform reader assignment, and utilize a shared memory buffer. LMDBIO-LMM adopts LMDB's API for accessing the database file efficiently. Moreover, LMDBIO-LMM abstracts parallel data reading from applications and provides a convenient C++ API that the applications can utilize to obtain the data for each process.

3.2.1 Detecting Colocated Processes

One of the first aspects that we need to solve in order to achieve localized `mmap` is to detect which processes reside on the same node—or more precisely, which processes share the same `mmap` page cache and bottom-half interrupt handlers.

Achieving this objective portably is hard. LMDBIO adopts a feature in MPI-3 to split a global MPI communicator into multiple local communicators (`MPI_Comm_split_type` with `MPI_COMM_TYPE_SHARED`). The general idea of this feature is to inform the user of the group of processes that are capable of allocating a shared memory buffer. In theory, the MPI implementation can provide any group of processes that are capable of creating a shared-memory buffer together. For example, this could be all the processes on the same non-uniform memory access (NUMA) domain or the same socket. In practice, however, this group is often the processes that reside on the same node and thus share the same page cache and bottom-half interrupt handlers. This gives us a semi-portable way to detect processes on the same node with the added convenience that in case the approach does not give the right set of processes, we can gracefully degrade performance rather than failing outright.

Once the communicators are set up, LMDBIO-LMM chooses one reader from each local

[7]Most supercomputers in the world already support MPI-3. The only notable exception to this claim is the IBM Blue Gene series of supercomputers that do not yet support MPI-3. However, these supercomputers are nearing their end of life, and the next generation of supercomputers from IBM do plan to support MPI-3 and later MPI standards.

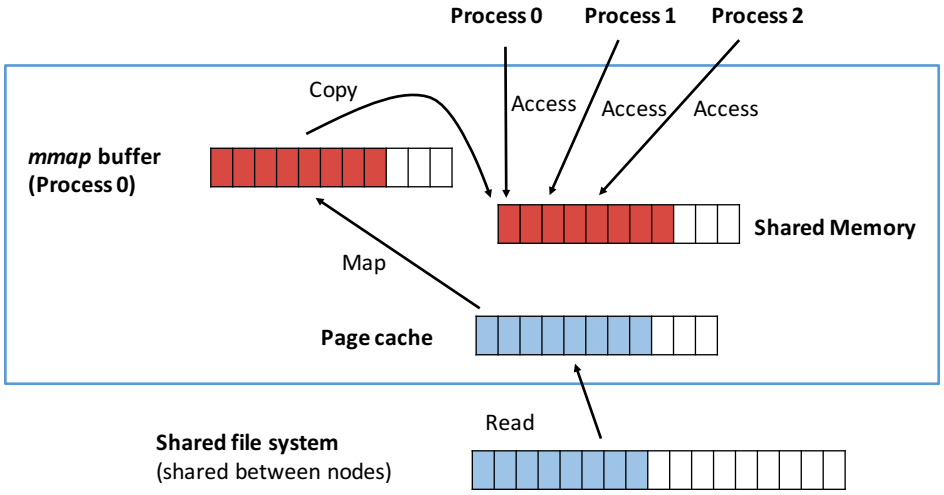

Figure 3.5: LMDBIO-LMM overview

group as the "root."

3.2.2 Inner Workings of LMDBIO-LMM

LMDBIO-LMM consists of two phases: an initialization phase and a data-reading phase.

Initialization phase: In the initialization phase, LMDBIO-LMM assigns one reader per node using the approach mentioned in Chapter 3.2.1. Then, each reader opens the LMDB database that internally maps the database to that process's virtual address space using `mmap`. All processes on the node also allocate a shared-memory buffer that is directly accessible by all of them.

Data-reading phase: In the data-reading phase (shown in Figure 3.5), each reader in LMDBIO-LMM (one process per node) reads B/R data samples (B is the batch size and R is the number of readers) from the database. The data is read from the filesystem to page cache and mapped to the virtual address space of the reader process. The reader process then copies the data to the shared memory buffer that was allocated during the initialization

phase in order to allow other processes to access this data. LMDBIO-LMM then synchronizes the processes within the local communicator (to ensure that the read has completed), after which the other processes on the node are allowed to access the shared memory buffer. Even though each process has full access to all of the shared memory buffer, LMDBIO-LMM internally limits such access so that each process can access only B/P samples of data (P is a total number of processes).

We note that several other approaches exist to implement shared memory between processes, for example, using `/dev/shm` or using `mmap`. These methods are portable on POSIX-based systems. However, we have chosen MPI-3's shared-memory implementation for two reasons: (1) the MPI implementation can choose the most suitable shared-memory model for a particular system including non-POSIX models such as XPMEM [171] and PiP [67], providing a minor performance advantage in some cases, and (2) we already use MPI for other communication in our framework, thus making MPI-3 shared memory a more natural model to be used in our framework.

3.3 Shortcomings of LMDBIO-LMM

Despite the various advantages of LMDBIO it still suffers from some shortcomings.

Serialized I/O: LMDBIO serializes I/O on each node, so only one process per node is doing the I/O rather than all processes. While this approach is beneficial for minimizing the amount of interprocess I/O contention that occurs with `mmap`, we lose the opportunity to maximize a read bandwidth of the file system. To utilize the I/O bandwidth more efficiently with multiple processes, we have to use other approaches, for example, direct I/O, in order to avoid the problems with `mmap`.

Buffer aliasing: In general, when a buffer is allocated, the allocated memory can contain pointers to other pieces of memory. Thus, any access on such memory could inadvertently modify other memory regions, a problem referred to as buffer aliasing. Most compilers are conservative in computing on aliased buffers since they need to be aware of such side effects and consequently generate less-optimized code. Malloc and malloc-like memory allocation calls are special in that they pass a special attribute to the compiler assuring it that any buffer allocated through malloc is not aliased. Thus, the compiler can perform more aggressive optimizations on this buffer, leading to better performance. Unfortunately, this "no aliasing" attribute can be passed to the compiler only when the return value of the allocation function is the memory buffer, not when the memory buffer is a function parameter. The MPI-3 shared-memory buffer allocation function misses this optimization opportunity: the shared-memory buffer that is allocated is not the return value of the function but, rather, a function parameter. This flaw in the MPI-3 interface design can cause degradation in the compiler optimization. The issue can be worked around by using `restrict` pointers to access the buffer, which provides the compiler with equivalent information as an unaliased buffer, thus achieving the same level of performance. However, this is an extra step that the application needs to be aware of.

TLB misses: In LMDBIO, the data samples are copied from the `mmap` buffer into a shared memory buffer. In traditional Caffe, this data is copied into a regular malloc buffer. Unfortunately, malloc buffers and shared-memory buffers differ significantly in the way the OS assigns physical pages to them. Buffers allocated with malloc use large (2 MB) pages on most processors, whereas shared-memory buffers use regular (4 KB) pages. Thus, computing directly on this buffer might cause a larger number of TLB misses when using shared memory than when using malloc. In Caffe, we are not affected by this issue because of a second transformation that copies the data again into another malloc buffer before any core

computation is done. Thus, the core computation itself does not suffer from the increased TLB misses. Nevertheless, this is an issue that future variants of Caffe need to be aware of.

3.4 LMDBIO-LMM Experiments and Results

In this section, we compare the performance of LMDBIO-LMM with that of LMDB. We split the experiments into two parts: (1) Evaluating the performance of LMDB and LMDBIO-LMM using simple microbenchmarks. The purpose of this evaluation is to understand the benefits and shortcomings of each optimization without diluting the results with other computation that would happen in a typical DL application. (2) Evaluating the performance of LMDB and LMDBIO-LMM using Caffe. The purpose of this evaluation is to understand the impact of LMDBIO on the overall performance of the Caffe DL framework on real datasets. Our experiments use the datasets, neural networks, and supercomputer systems described in Chapter 3.1.1.

3.4.1 Microbenchmark Evaluation and Analysis

We evaluate the performance of LMDB and LMDBIO-LMM using the microbenchmark that emulates the I/O behavior of Caffe. Our microbenchmark is designed to use LMDB or LMDBIO to perform file I/O. It performs iterative file I/O, similar to what Caffe would, but it does not perform any of the computation associated with DNN training. The experiment in this section is performed on Bebop, with 9.4 million images of the CIFAR10-Large database; the batch size is set to 18,432 images.

Figure 3.6(a) shows a comparison of the read performance of LMDB and LMDBIO-LMM. We see that LMDBIO-LMM outperforms LMDB by up to 43.77-fold when the number of

cores is smaller than or equal to 1,152. This improvement in performance is attributed to reduced interprocess contention. For large numbers of cores, LMDBIO-LMM performs slightly worse than LMDB because we used a single root process on each node in all our experiments for consistency. Increasing to two root processes per node, when running on a large number of cores, improves the LMDBIO-LMM performance enough to address this degradation, although those numbers are not shown in this graph.

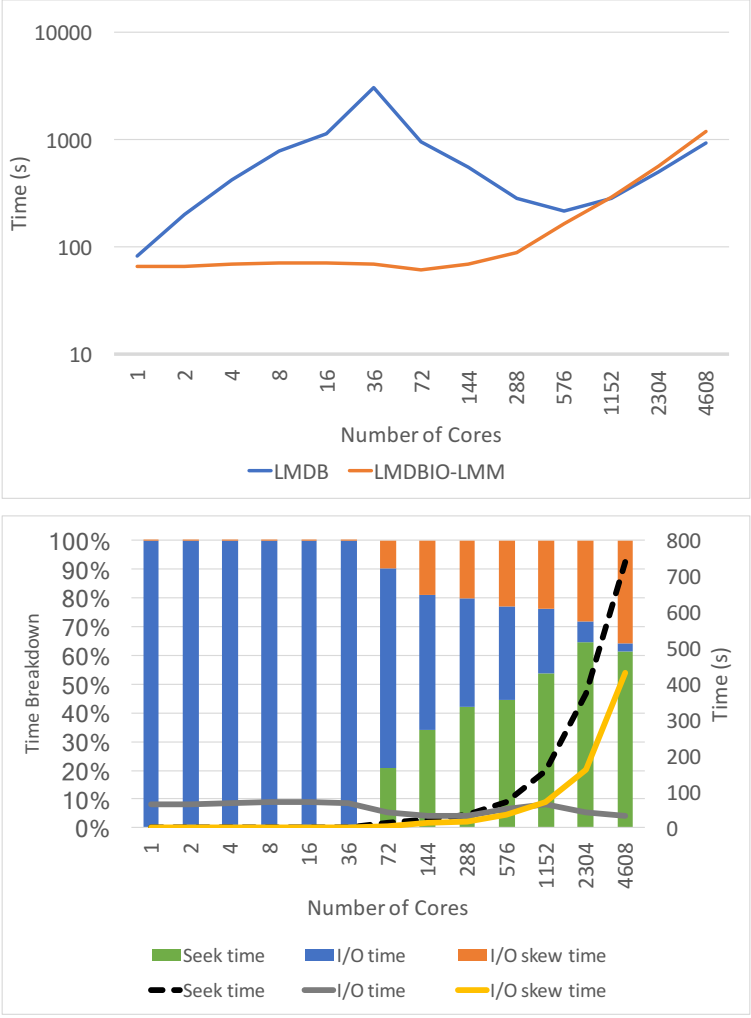

Figure 3.6: LMDBIO-LMM performance analysis: (a) read performance compared with LMDB; (b) total read time breakdown

Figure 3.6(b) shows the breakdown of the total read time, divided into the seek time (which

is still sequential), the I/O time, and the I/O skew time. From the graph, we observe that although the I/O time itself is fairly small, a significant amount of time is spent in the sequential seek and in the skew among processes where some processes are waiting for other processes to catch up. These two are related. The skew is caused by the data-seeking process in LMDBIO-LMM.

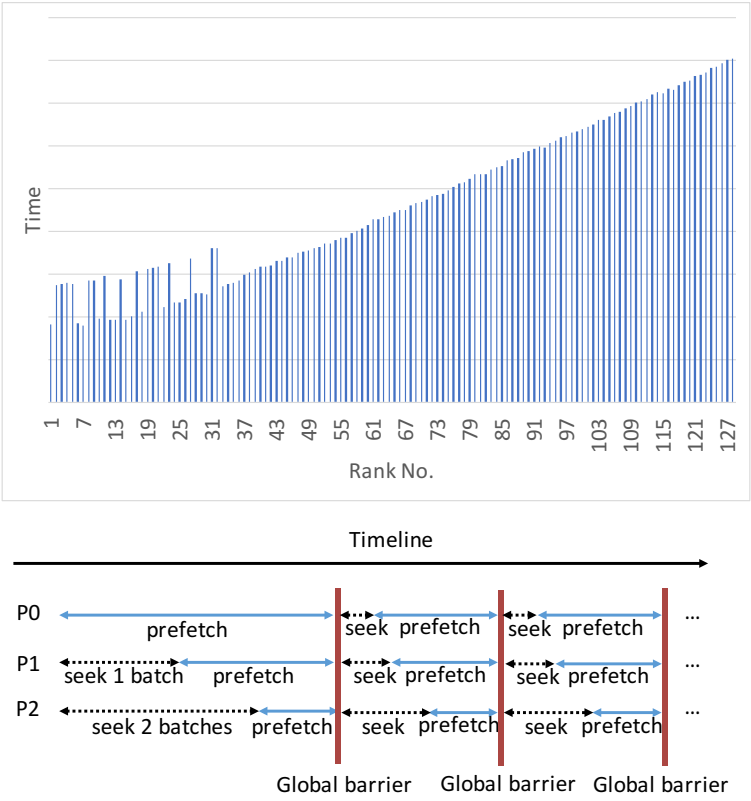

Figure 3.7: LMDBIO-LMM performance analysis: (a) seek time vs. reader's rank number; (b) `mmap`'s data prefetching and data seeking

To further understand this, we perform an additional analysis using a small benchmark that contains only the seek part and plot it against the root process's MPI rank, as shown in Figure 3.7(a). We observe that the seek time increases with the process rank. This phenomenon is a subtle outcome of the data prefetching that is performed within `mmap`, as demonstrated in Figure 3.7(b). Specifically, after opening a database, each reader process

will initialize the cursor by seeking to its corresponding starting location in the database. The amount of seek that is performed by each reader is $rank \times B$ ($rank$ denotes the reader process's rank, and B denotes the batch size), which is not uniform among the different processes. Because of the bulk synchronous nature of the computation, however, some processes end up waiting in the synchronization longer than others. The processes that wait in the synchronization longer thus have a better opportunity to prefetch data that they would need in the next iteration. For instance, $P0$ spends a large part of its time prefetching data for the next iteration, while $P2$ gets very little time to prefetch. This prefetching, in turn, helps $P0$ with its seek in the next iteration, thus causing it to complete faster than the other processes in that iteration as well, so the cycle continues and results in large skew.

To analyze the reduced contention, we show in Figure 3.8 the number of context switches that occur with LMDBIO-LMM compared with LMDB. We can see that in some cases LMDBIO-LMM achieves more than an 83-fold reduction in the number of context switches compared with LMDB, thus demonstrating that our technique to localize `mmap` can significantly reduce interprocess contention.

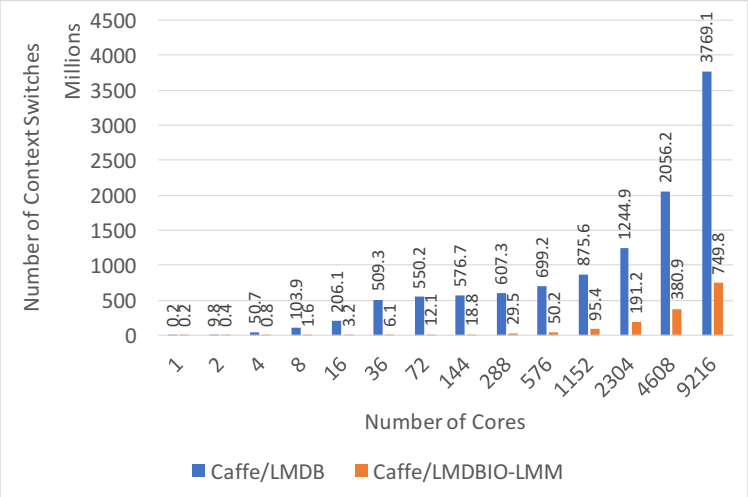

Figure 3.8: LMDBIO-LMM performance analysis: context switches compared with LMDB

3.4.2 Evaluation of Caffe Deep Learning Training

In this section, we use the real DL framework, Caffe, to evaluate the performance of I/O subsystems. The results of LMDB and LMDBIO in this section are denoted as Caffe/LMDB and Caffe/LMDBIO, respectively. We perform our experiments on two platforms, Blues and Bebop, using all three datasets.

Evaluation Results on Blues

We train CIFAR10-Large and ImageNet on Blues by using a batch size of 4,096 for both datasets. CIFAR10-Large is trained for 1,024 iterations (4 million images), while ImageNet is trained for 32 iterations (128K images). We scale the training from 1 to 512 cores (32 nodes).

Figure 3.9(a) shows the strong-scaling results for training Caffe with the CIFAR10-Large dataset on Blues. Caffe/LMDBIO-LMM outperforms Caffe/LMDB in all cases, achieving 1.21-fold improvements on 512 cores. For ImageNet (Figure 3.9(b)), the improvement reaches up to 7-fold, which is attributed to the reduced interprocess contention in LMDBIO-LMM.

Evaluation Results on Bebop

We train CIFAR10-Large and ImageNet-Large on Bebop. The batch size used in the experiment for both datasets is 18,432. CIFAR10-Large is trained for 512 iterations (9 million images), and ImageNet-Large is trained for 128 iterations (2 million images).

Here, we show both strong scaling results and performance breakdown of both datasets.

Figure 3.10(a) compares the performance of the original Caffe/LMDB with that of Caffe/LMDBIO for the CIFAR10-Large dataset. The general trend that we notice is that

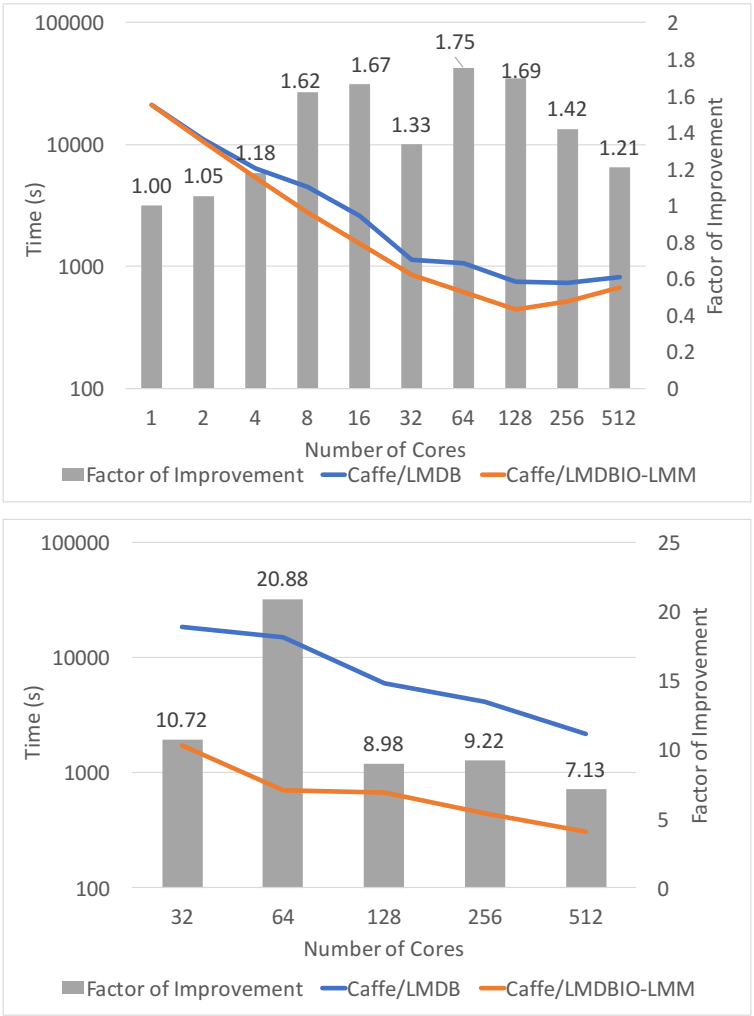

Figure 3.9: Caffe/LMDBIO-LMM strong scaling on Blues using (a) CIFAR10-Large; (b) ImageNet

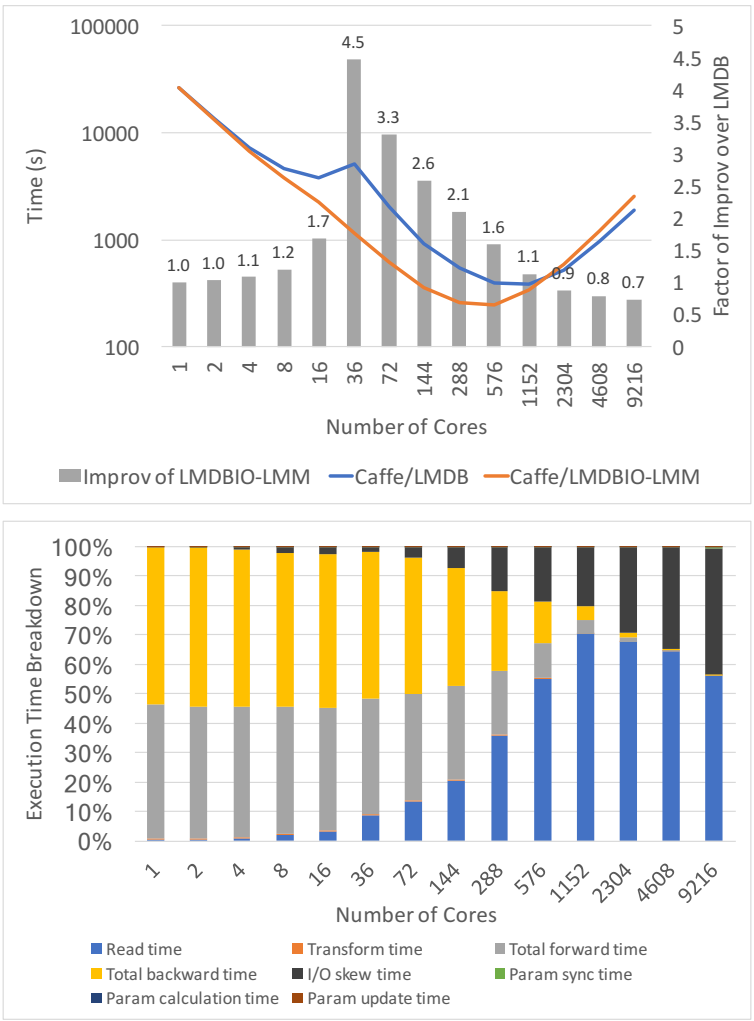

Figure 3.10: Caffe/LMDBIO-LMM strong scaling on Bebop using CIFAR10-Large: (a) scaling results; (b) performance breakdown

Caffe/LMDBIO performs better than Caffe/LMDB by up to a factor of 4.5. Figure 3.10(b) shows the breakdown of performance for Caffe/LMDBIO where we notice that time taken by the file I/O (represented as "Read time" in the figure) has reduced significantly compared with that of the original Caffe/LMDB implementation (shown in Figure 3.1(b)), specifically from 90% to 56% on 9,216 cores.

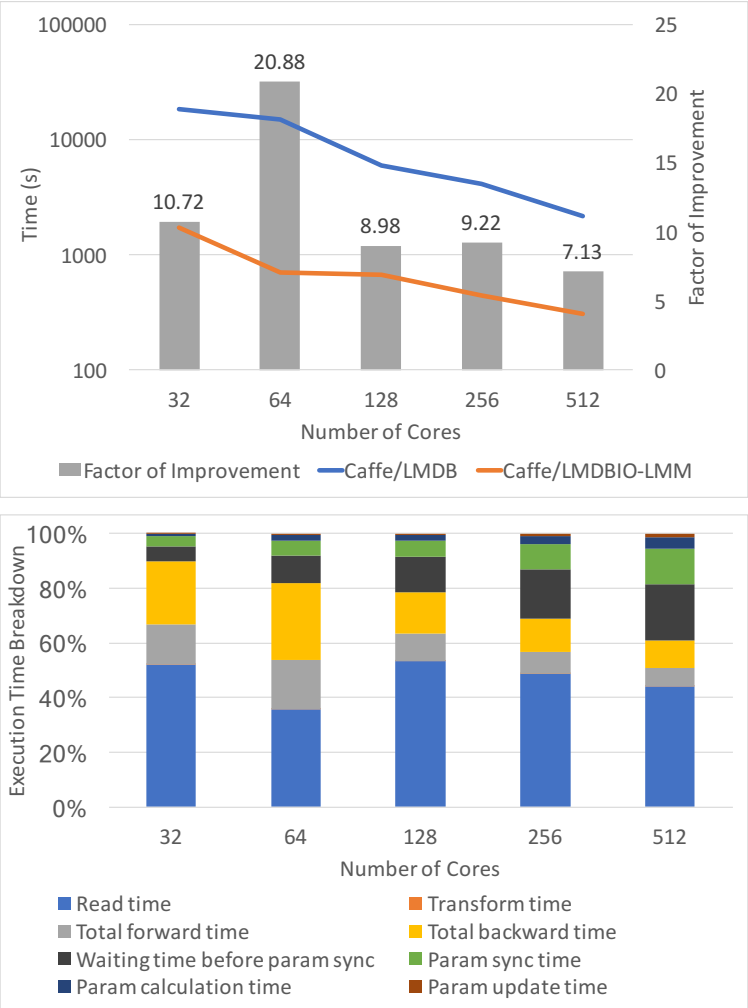

Figure 3.11: Caffe/LMDBIO-LMM strong scaling on Bebop using ImageNet-Large: (a) scaling results; (b) performance breakdown

We perform a similar analysis on the ImageNet-Large dataset, as shown in Figures 3.11(a) and 3.11(b). The general trend for ImageNet-Large is similar to that of CIFAR10-Large.

The performance breakdown graphs of both datasets complement with our findings in Chapter 3.4.1 that LMDBIO-LMM suffers from I/O skew. The results show that I/O skew takes up to 42% and 26% of the total execution time of CIFAR10-Large and ImageNet-Large, respectively.

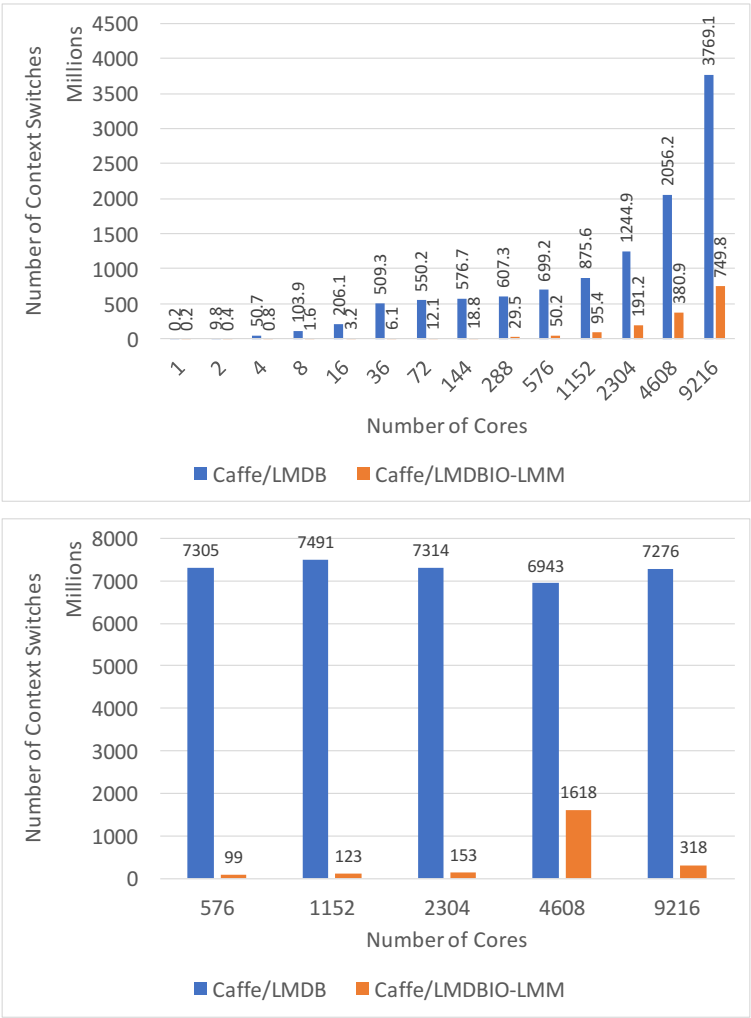

Figure 3.12: Caffe/LMDBIO-LMM context switches compared to Caffe/LMDB on Bebop using (a) CIFAR10-Large; (b) ImageNet-Large

Figure 3.12 shows the improvement of context switches for both datasets. We see that the number of context switches is significantly better for Caffe/LMDBIO-LMM compared with that of the original Caffe/LMDB. For the CIFAR10-Large dataset, LMDBIO-LMM reduces

the number of context switches by nearly 5-fold. For ImageNet-Large, the improvement is even better, with close to a 22-fold reduction in the number of context switches.

This improvement is expected. Since LMDBIO has a single process performing `mmap`, it ensures that no contention occurs between `mmap` calls performed by multiple processes. This serialization reduces the number of unnecessary wakeups created by the interrupt handler, thus reducing the number of context switches.

3.5 Chapter Summary

In this chapter, we presented LMDBIO-LMM, an intra-node I/O optimization for scalable DL, that is based on LMDB. We first performed a detailed analysis of file I/O in Caffe, showcased and discussed the interprocess contention problem caused by `mmap` when using multiple readers. We then presented LMDBIO-LMM, which alleviates the presented problem by localizing `mmap`. We presented experimental results with three datasets and two neural network models on two platforms, demonstrating nearly a 20-fold improvement in the overall performance of Caffe in some cases.

We note that our work is not fully completed yet. File I/O still takes a large portion of the overall time even in LMDBIO-LMM. Nevertheless, the proposed approach is still a significant step toward improving I/O performance for DL. Moreover, this improvement is despite the reduced I/O parallelism and the extra data copy that we perform within Caffe/LMDBIO-LMM.

Chapter 4

Inter-node File I/O Optimization via Speculative Parallel I/O

The problem of LMDB discussed in this chapter is its sequential database access limitation. LMDB database format, B+ tree, does not allow random database accesses. In our investigation, we found that this limitation causes a large amount of redundant data reading among readers. Despite redundant data reading, this causes skew in file I/O as different processes perform a different amount of work, which can severely degrade the overall progress of a parallel application.

To tackle the problem, we propose **LMDBIO-LMM-DM**—distributed memory file I/O optimization, which is a successive optimization of LMDBIO-LMM for distributed data reading. LMDBIO-LMM-DM leverages two techniques, speculative data reading, and reader collaboration, in order to reduce redundant and imbalanced data reading as well as enhance file I/O parallelism.

We evaluate the performance of LMDBIO-LMM-DM and compare it to that of LMDB and LMDBIO-LMM using a microbenchmark and the actual benchmark to train multiple datasets on two different distributed memory platforms. Our results show that LMDBIO-LMM-DM consistently outperforms other frameworks. We also provide the discussion on the accuracy of our speculative data reading at the end of this chapter.

4.1 Analysis of LMDB Sequential Data Access Restriction

As mentioned in Chapter 2.4.2, LMDB does not support random database accesses; that is, the LMDB database can be accessed only sequentially. This is a significant issue for parallel DL because each process needs to read and process a different subset of data (typically interleaved with the data required by other processes). Thus, each process needs to start from the root of the database and parse through (and skip) all the intermediate records until it reaches the desired record that it wants to process. We refer to this operation as the "sequential seek" operation, although unlike a traditional UNIX seek operation, it is not possible to directly jump to an arbitrary page without risk of accidentally reaching an overflow page that contains no information on how to go to the next or leaf node.

While traversing through the tree nodes, the header on each node is read to obtain a pointer to the next record location. To do so, the page containing the header needs to be loaded into memory. Since the header itself is much smaller than the physical page size, the header page usually contains additional information that needs to be loaded into memory even when it does not need to be accessed.

The data access pattern when using LMDB in parallel is demonstrated in Figure 4.1. Suppose four readers ($P0$ - $P3$) need to read a different portion of the database ($D0$ - $D3$) from the filesystem to memory. When $P0$ reads $D0$, it reads both the headers and the actual content. In this case, $P0$ does not read any extra data. In order to read $D1$, however, $P1$ has to seek through all of the branch and leaf nodes in the $D0$ portion of the database before it gets to the $D1$ portion. From the figure, we notice that the amount of extra data read increases with the process count, where in this case $P3$ reads the most extra data. With this data access model, in the worst case a process could end up reading a total of $R \times B$ bytes, where

R is a total number of readers and B is a size of an individual data portion.

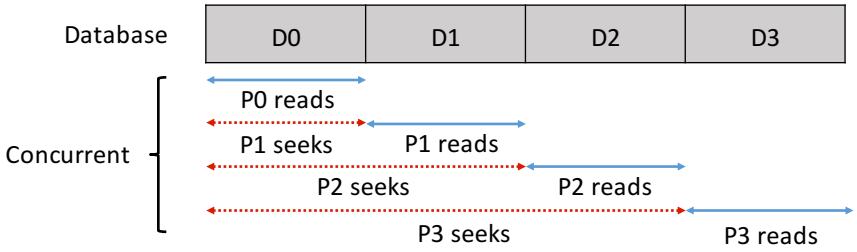

Figure 4.1: LMDB redundant data movement

Apart from a large amount of redundant file I/O, this data access also causes large imbalance and skew in data reading because each process reads a different number of bytes. Such a load imbalance can cause processes to stay idle at a process synchronization point (e.g., parameter synchronization in DL) waiting for the last process to finish its task. This can severely degrade the overall progress of a parallel application.

4.1.1 Analysis of Amount of Extra Data Fetched

In this section, we perform a more profound analysis to understand the amount of data that is fetched to each node with Caffe/LMDBIO-LMM. To perform our measurements, we modify Caffe to initially memory-protect all of the memory-mapped database. When a page in the database gets accessed, it triggers a page-fault handler, which we catch to measure the amount of data that would be fetched by the filesystem. Once the page is touched, it is unprotected, so any future accesses to the page do not raise additional page faults. We note that the most accurate way to measure the exact number of bytes read is to profile the filesystem driver performing the operations. The GPFS file I/O framework, however, is closed source, making that an impossible option. Thus, the method we suggested above is an approximation of the behavior to estimate the total data read, which we believe is fairly accurate. There are two aspects this approximation does not consider.

1. For each page that generates a page fault, at least one page of data would be fetched from the filesystem. Some filesystems perform prefetching of additional data during page faults. Unfortunately, since GPFS is closed-source, we could not measure the exact amount of such additional prefetch data. However, since our data access is contiguous, we believe that this additional unaccounted data would be small.

2. The filesystem has a page cache that is limited. Thus, if the processes access too many pages, eventually they will run out of cache space and start swapping out pages that are already in memory and fetch them again when needed. While, in theory, it is possible for this to happen, we argue that it is improbable in practice (in our particular use case) because of two reasons. First, data access in LMDB is mostly (though not entirely) sequential. Most accesses are to a few of the most recently, and even if an existing page gets swapped out, it is highly unlikely that it would get fetched back in. Second, on modern OSs, this limit is most of the memory available on the system. In our experiments, the amount of data read per node is much smaller than this limit.

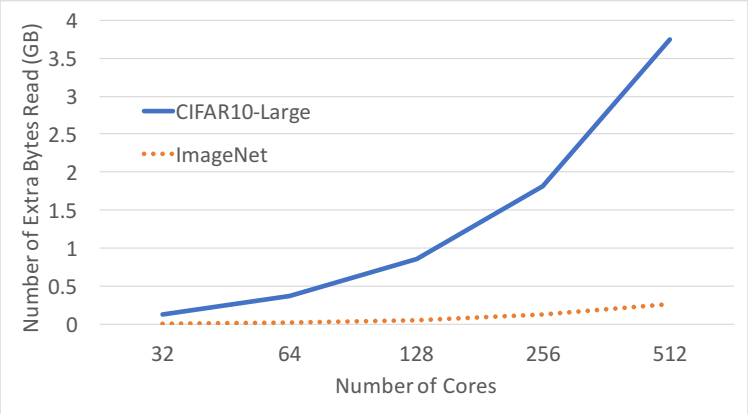

Figure 4.2: Caffe/LMDBIO-LMM extra bytes read

We collect the results by training CIFAR10-Large and ImageNet for 1,024 iterations and 32 iterations, respectively. Both experiments are executed using Caffe with LMDBIO-LMM on Blues.

Figure 4.2 shows the number of "extra" bytes read by Caffe/LMDBIO-LMM for the CIFAR10-Large and ImageNet datasets—that is, how many additional bytes are read by the different processes apart from the actual data that they need for their processing.

We make two observations based on this figure. First, the number of additional bytes increases with the number of cores for both datasets, reaching 4 GB in some cases. In fact, the increase is almost linear with the number of cores. This increase is due primarily to the redundancy in the data read as we increase the number of cores. Specifically, a process on each core needs to seek through the data read by all of the previous processes.

Our second observation is that the increase for the ImageNet dataset is much smaller than that of the CIFAR10-Large dataset. This is also expected, although the reason is more subtle. As a process seeks through the dataset to reach its relevant portion of the database, it needs to read the appropriate headers of the database pages. As discussed in Section 2.4.2, these pages are branch and leaf pages. In the CIFAR10-Large dataset, each data sample is approximately 3 KB. Thus, the header and the data sample reside on the same physical page. Consequently, reading the header would load the entire data sample into memory, causing a large amount of additional data to be fetched into memory. In the ImageNet dataset, on the other hand, each data sample is approximately 192 KB and thus takes around 48 pages to store. Therefore, the header page is encountered fewer times in ImageNet if the total dataset size of both datasets is approximately the same; that is, there is one header for every physical page in the CIFAR10-Large dataset, whereas there is one header for every 48 physical pages in the ImageNet dataset. This results in fewer additional bytes read for the ImageNet dataset.

4.2 Design and Implementation of LMDBIO-LMM-DM: Distributed Memory File I/O Optimization

In this section, we present details of the design and implementation of LMDBIO-LMM-DM. The LMDBIO-LMM-DM software itself is an extension of the original LMDBIO-LMM software and has been developed on top of the same code base. Details of LMDBIO-LMM can be found in Chapter 3.

As discussed in Chapter 4.1, one of the primary reasons for the performance loss in file I/O with Caffe/LMDB and Caffe/LMDBIO-LMM is the redundant file I/O by the different processes. To solve this issue, we propose a two-step approach. In the first step, described in Chapter 4.2.1, we present an approach where each process reads exactly the data that it needs to process, although it does so by serializing I/O across the different processes. In the second step, described in Chapter 4.2.2, we present an approach for estimating what data pages each process will eventually need and speculatively performing parallel I/O to regain most of the performance lost because of the I/O serialization described in the first step.

4.2.1 Serializing I/O Using a Portable Cursor Representation

Here, our goal is to ensure that each process reads only the data that it needs to process. In other words, no additional data is read at seek time. To do so, each process must first read the data that it needs to process and then pass to the next process the information about the location where it stopped. The general model we want to follow is illustrated in Figure 4.3. In the figure, $P1$ cannot start reading data $D1$ until $P0$ finishes reading $D0$ and sends the starting point of $D1$ (i.e., the cursor) to it. Executing this in practice, however, has a few complications that we discuss in this section.

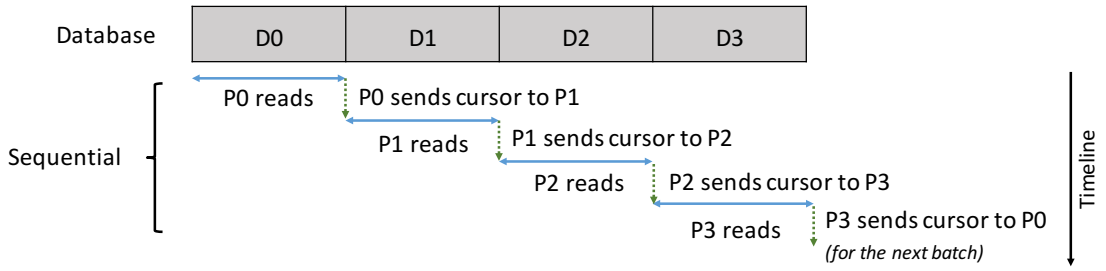

Figure 4.3: LMDBIO-LMM-DM design: sequential I/O and cursor handoff

As described in Chapter 2.4.2, since LMDB uses a B+ tree to represent its data elements, the position indicator for a record within the B+ tree is not a simple offset from the start of the file, but rather a more complex data structure which LMDB refers to as a cursor. The cursor includes information about the record it points to. But it also includes other information such as the path of the record's parent branch nodes, a pointer to the page header containing the record, and information about the access flags of the particular record being pointed to. Unfortunately, the cursor data structure itself is not portable across different processes since it contains information represented as pointers within the B+ tree that is relevant only within the virtual address space of the original process that created the cursor. Luckily, all the pointers contained within this structure point to locations within the B+ tree.

In order to serialize the cursor into a format that is portable across different processes, the simplest model that we envision is that of a symmetric address space. That is if we can ensure that all processes can memory-map the database into exactly the same virtual address location on all processes, any pointers that point to locations within the B+ tree would be portable across the different processes, thus making it possible to serialize the cursor to a portable format. To achieve this, we use the following algorithm. The first reader process randomly picks a virtual address location from its 64-bit address space and tries to memory-map the database to this memory location. If it is successful, it broadcasts this address to the remaining reader processes. Each of the remaining reader processes tries to memory-

map the database file at the exact same memory location. Each process indicates whether it was successful or not within an `MPI_Allreduce` operation where all processes try to come to a consensus. If everyone was successful, the database is now mapped to the same virtual address location on all processes. If at least one of the processes was not successful, all processes unmap their database and try again. This process is repeated for a few iterations.

In theory, it is possible to find no virtual address location that can be symmetrically used across all processes. However, given that most of the 64-bit address space is typically unused on any given process, in practice, we can find a symmetrical address space in 1–2 attempts with the algorithm described above. In the worst case, if we are not able to find a symmetrical address space after a few attempts, we abandon this optimization and fall back to the approach used by the original LMDBIO-LMM.

An alternative approach to achieve the same outcome would be to modify the LMDB implementation such that the user could pass in a pointer offset that would be used for the database parsing. Such an approach would be equally effective, although we feel that the symmetric memory allocation technique that we use in this work is a less intrusive (to the LMDB implementation) and generally more elegant solution to the problem.

Once the database is mapped to the symmetrical address space, the actual serialization of the cursor itself is mostly trivial. The internal content of the cursor data structure is copied into a memory buffer that can be sent to the other processes by using MPI send/recv.

4.2.2 Speculative Parallel I/O

The first step of our algorithm, discussed in Chapter 4.2.1, provides a portable solution to pass the location information within the database to other processes. However, the approach described there comes at the cost of serialization in file I/O. That is, only one process is

actively reading data at any given point of time. This is inefficient on most parallel filesystems where multiple processes need to be performing I/O to achieve the best performance.

Here, we discuss the second step of our algorithm that tries to estimate what data needs to be processed by a given process and speculatively performs parallel I/O on that data (illustrated in Figure 4.4). To do this, we must first estimate what part of the database we need to fetch to memory. This is a complex task since the structure of the B+ tree is not always straightforward. Depending on how many branch nodes are used and how many records each branch node points to, estimating which physical pages each process would need to access is nontrivial.

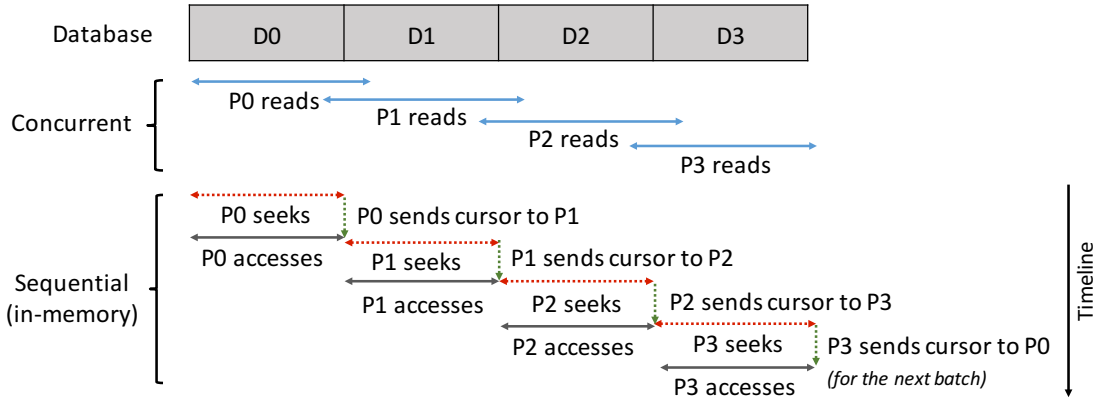

Figure 4.4: LMDBIO-LMM-DM design: speculative parallel I/O and in-memory sequential seek

In our approach, we assume that the sizes of all records are roughly the same, which is a reasonably safe assumption to make for most DL frameworks because of the way the input data samples are handled. Each reader process reads the first data record in the database file to retrieve the record's size. The readers use the obtained size information along with the number of records that they will read (i.e., a fraction of the batch) to estimate the number of pages to be fetched. For instance, the size of each sample in the CIFAR10-Large dataset is approximately 3 KB. For I/O efficiency reasons, LMDB pads the data to ensure that each

record occupies one page (4 KB). Therefore, the number of speculative pages for n data records is $n \times 1$ pages. The read offset of each reader is calculated in the same fashion.

Once we guess what pages we need to process, each process "touches" the appropriate pages in the memory-mapped database file, thus forcing the filesystem to fetch those pages to memory. This step is done in parallel on all processes. Once the data has been fetched to memory, we perform the sequential seek process described in Chapter 4.2.1 to find the starting point of the data batch for the next reader. We expect, however, that this sequential seek process accesses only or mostly pages that are already in memory and thus will be quick compared with the file I/O itself. Once the seek is done, and the reader successfully sends the starting location to the corresponding process, the reader can perform the actual data processing.

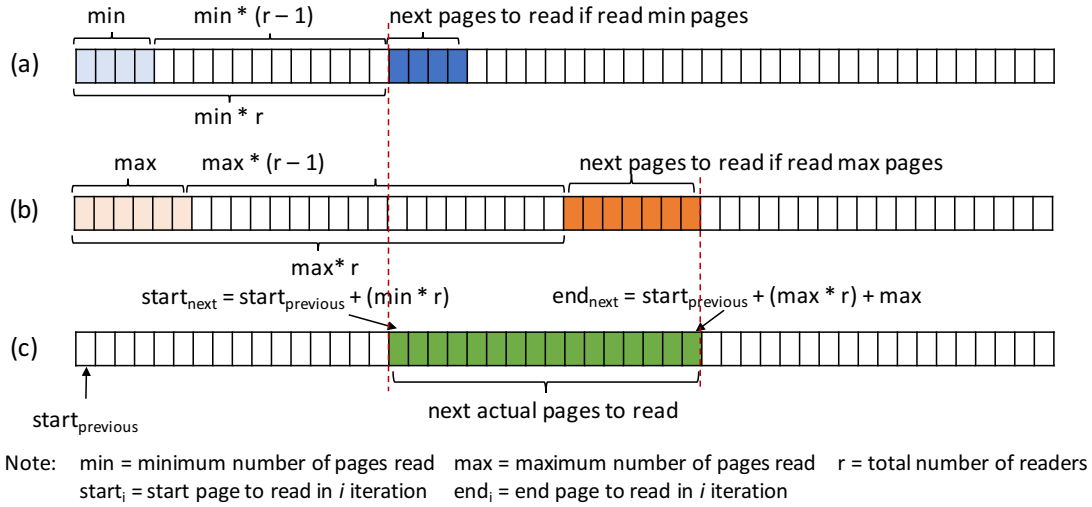

Figure 4.5: Pages accessed by a reader process using the history-based speculative read approach: (a) pages accessed if the minimum number of pages are read; (b) pages accessed if the maximum number of pages are read; (c) pages accessed in the actual case

In the next iterations, we use a history-based approach to correct the speculative read pages. Each reader process maintains a *minimum* (denoted *min*) and *maximum* (denoted *max*) number of pages that it has accessed so far. The key idea of the correction method is to

expand boundaries of speculative read pages based on *min* and *max* as shown in Figure 4.5. Figure 4.5(a) and Figure 4.5(b) illustrate the pages to read in the case that *min* and *max* number of pages are read in the next iteration, respectively. To minimize the amount of missed pages (i.e., pages that are supposed to be read but are not read), the reader process speculatively reads all pages between the *min* and *max* boundaries as demonstrated in Figure 4.5(c). We expect that over a few iterations, we get a fairly accurate picture of the branch structure of the database file that will allow us to estimate more precisely.

We note that the proposed history-based speculative I/O technique can be used in conjunction with other in-memory data shuffling approaches, where shuffling is done after the I/O has completed. In other words, as long as data I/O is structured and iterative (which is true for most DL frameworks), one can take advantage of the proposed history-based speculative I/O technique.

4.3 LMDBIO-LMM-DM Experiments and Results

Evaluation of LMDBIO-LMM-DM is divided into three subsections: (1) Evaluating the performance of LMDBIO-LMM-DM against LMDB and LMDBIO-LMM using a microbenchmark. (2) Evaluating the performance of LMDBIO-LMM-DM against the other two I/O subsystems using the Caffe DL framework. (3) Evaluating the accuracy of our speculative parallel I/O of LMDBIO-LMM-DM. Our experiments use the datasets, neural networks, and supercomputer systems described in Chapter 3.1.1.

4.3.1 Microbenchmark Evaluation and Analysis

The microbenchmark used in this section is the same as the one used in Chapter 3.4.1. The microbenchmark performs iterative file I/O using LMDB or LMDBIO; however, it excludes the training part. The experiment is performed on Bebop using the CIFAR10-Large dataset with a batch size of 18,432 images (9.4 million images).

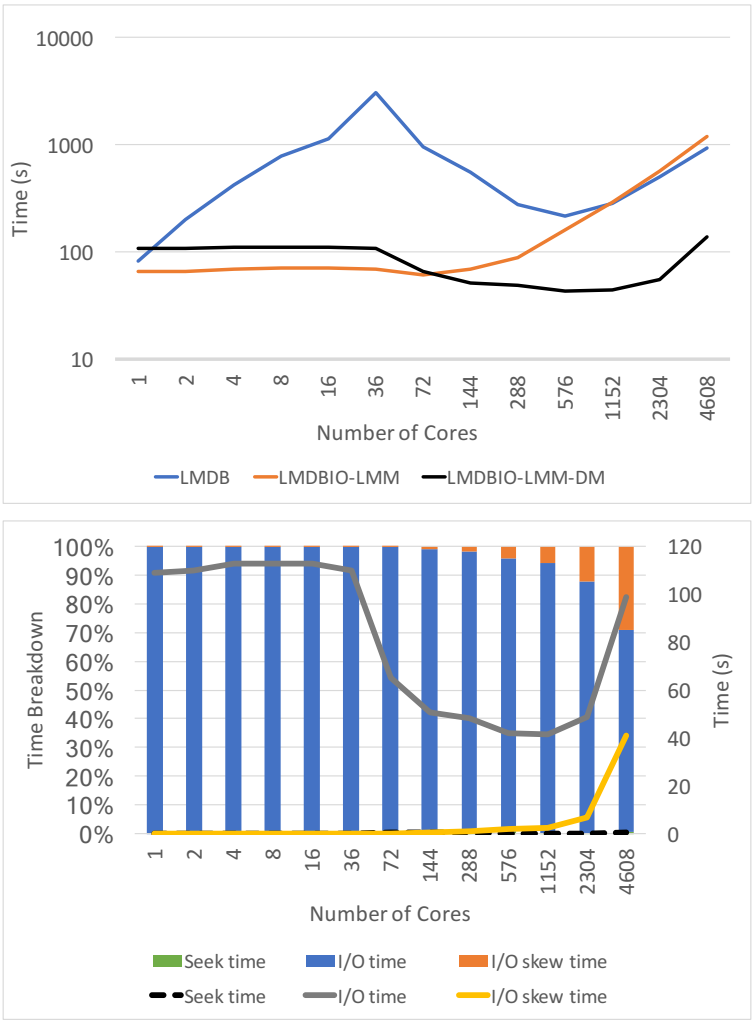

Figure 4.6: LMDBIO-LMM-DM performance analysis: (a) read performance compared with LMDB and LMDBIO-LMM; (b) total read time breakdown

Figure 4.6(a) compares the performance of LMDBIO-LMM-DM with that of LMDB and LMDBIO-LMM. On a single node, LMDBIO-LMM-DM does not achieve any performance

improvement compared with LMDBIO-LMM because it utilizes the same general principle as LMDBIO-LMM to avoid interprocess contention. In fact, the additional file I/O performed by LMDBIO-LMM-DM hurts performance somewhat, causing it to achieve slightly worse performance compared with LMDBIO-LMM. When using multiple nodes, however, LMDBIO-LMM-DM performs better than both LMDB and LMDBIO-LMM, outperforming LMDB by 6.7-fold on 4,608 cores. This improvement is attributed primarily to the reduction in redundant data read during the seek and to the speculative parallel I/O.

We study the breakdown of the LMDBIO-LMM-DM read time in Figure 4.6(b), which shows that the data seek in LMDBIO-LMM-DM is highly efficient compared with that of LMDB and LMDBIO-LMM. The seek in LMDBIO-LMM-DM takes less than 1% of the read time, compared with LMDBIO-LMM, which spends nearly 60% of the read time in seek. The better performance with LMDBIO-LMM-DM is mainly because the seek is performed in memory. In fact, in some cases, LMDBIO-LMM-DM improves the seek time by nearly 1,741-fold compared with LMDBIO-LMM.

4.3.2 Evaluation of Caffe Deep Learning Training

In this section, we evaluate the performance of our optimization using Caffe. The experiments are run on both Blues and Bebop. Each platform uses different datasets in the evaluation.

Evaluation Results on Blues

We evaluate Caffe/LMDBIO-LMM-DM against Caffe/LMDB and Caffe/LMDBIO-LMM using two sets of datasets, which are CIFAR10-Large (trained for 1,024 iterations) and ImageNet (trained for 32 iterations). Both experiments use a batch size of 4,096. We scale the training from 1 to 512 cores (32 nodes).

Figure 4.7(a) compares the performance of Caffe/LMDBIO-LMM-DM with that of Caffe/LMDBIO-LMM and Caffe/LMDB for the CIFAR10-Large dataset. Caffe/LMDBIO-LMM-DM performs better than Caffe/LMDBIO-LMM by around 1.87-fold and better than Caffe/LMDB by around 2.65-fold. The primary improvement in performance for LMDBIO-LMM-DM is attributed to the reduced data movement compared with that of LMDBIO-LMM and LMDB. Even though LMDBIO-LMM-DM introduces additional serialization in the file I/O path compared with LMDBIO-LMM and LMDB, the impact of this serialization is minimal because of the speculative parallel I/O that it performs.

With LMDBIO-LMM-DM, most of the file I/O is parallelized across processes, and each process reads mostly distinct parts of the database. Some serialization still exists in the cursor propagation across the different processes; but since that propagation is done almost entirely in memory without requiring file I/O, the impact of such serialization is minimal. This helps reduce the skew significantly.

We performed a similar analysis on the ImageNet dataset, as shown in Figure 4.7(b). The general trend for ImageNet is similar to that of CIFAR10-Large.

Evaluation Results on Bebop

In this section, we use two datasets to evaluate Caffe/LMDBIO-LMM-DM against Caffe/LMDB and Caffe/LMDBIO-LMM. We train CIFAR10-Large and ImageNet-Large for 512 iterations and 128 iterations, respectively. Both datasets use a batch size of 18,432. We scale our experiments from 1 core to 9,216 cores.

Figure 4.8(a) shows performance comparison of Caffe/LMDB, Caffe/LMDBIO-LMM, and Caffe/LMDBIO-LMM-DM using CIFAR10-Large. Similar to the results on Blues, Caffe/LMDBIO-LMM-DM begins to outperform other I/O subsystems when scaled on to

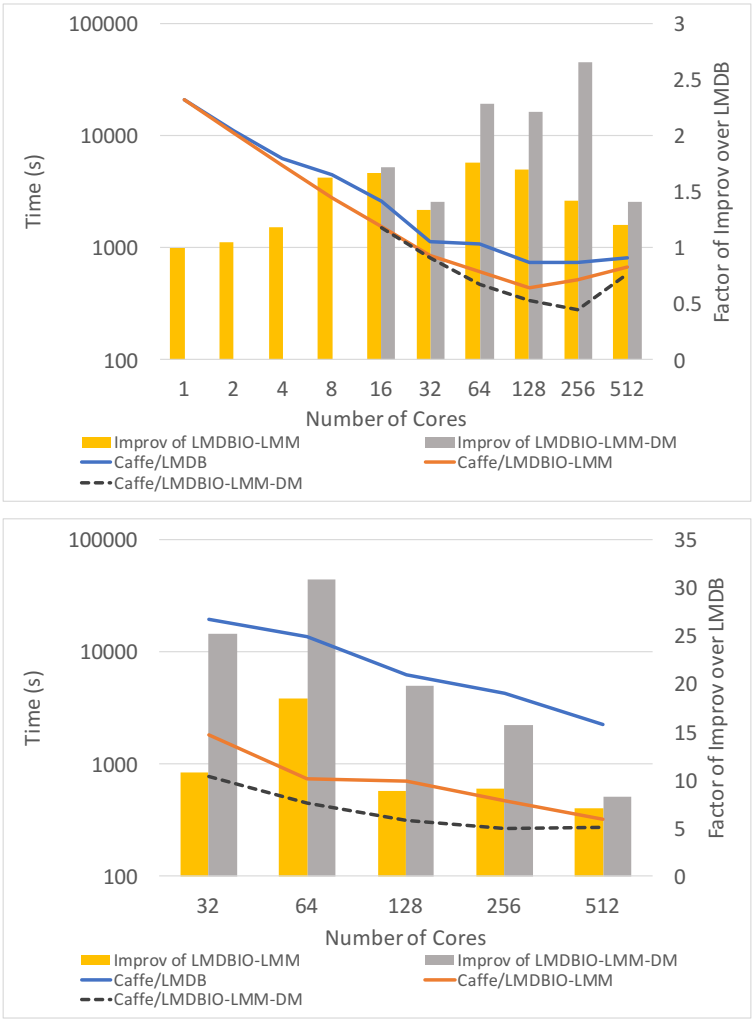

Figure 4.7: Caffe/LMDBIO-LMM-DM strong scaling on Blues using (a) CIFAR10-Large; (b) ImageNet

multiple nodes. In the best case, Caffe/LMDBIO-LMM-DM achieves 9.2-fold improvement from Caffe/LMDB. Figure 4.8(b) shows that the improvement of LMDBIO-LMM-DM from LMDBIO-LMM is subject to the smaller amount of I/O skew. The I/O skew time of LMDBIO-LMM-DM is less than 10% from 1 to 4,608 cores and is 30% on 9,216 cores, while that of Caffe/LMDBIO-LMM begins to be larger than 10% from 288 cores onward (see Figure 3.10(b)).

Figure 4.9 shows performance comparison of Caffe/LMDB, Caffe/LMDBIO-LMM, and

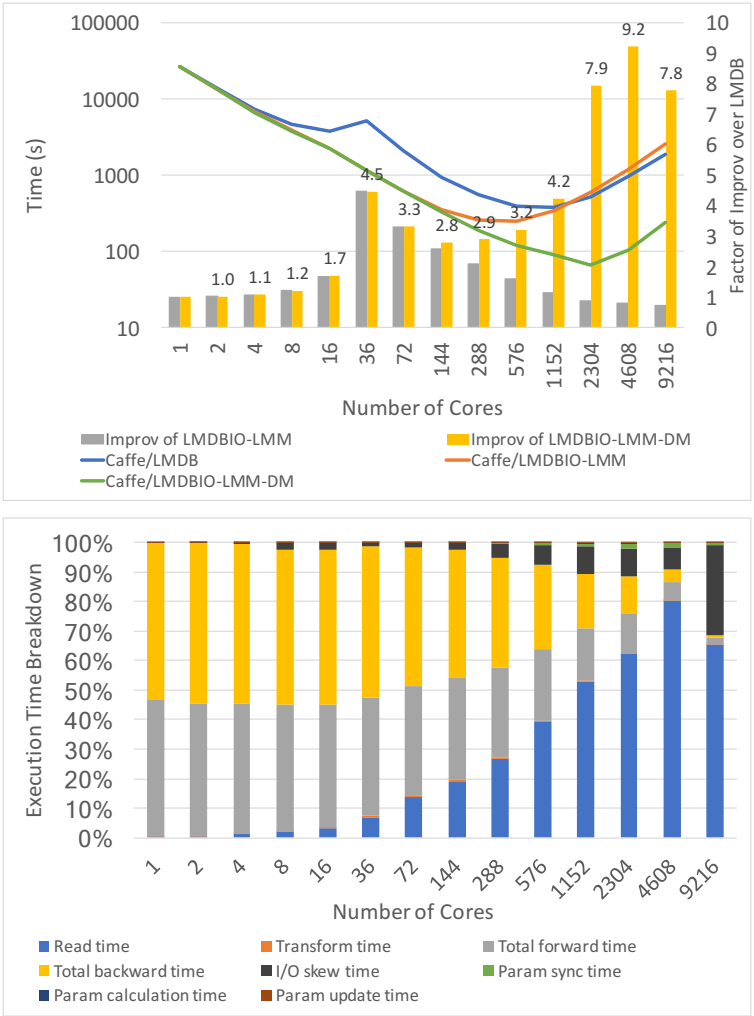

Figure 4.8: Caffe/LMDBIO-LMM-DM strong scaling on Bebop using CIFAR10-Large: (a) scaling results; (b) performance breakdown

Caffe/LMDBIO-LMM-DM using ImageNet-Large. From the results, Caffe/LMDBIO-LMM-DM outperforms Caffe/LMDB in all cases but performs similarly to Caffe/LMDBIO-LMM. The reason is that Caffe/LMDBIO-LMM-DM on Bebop suffers from the problem that we refer to as "I/O randomization", which we will further discuss it in Chapter 5.1.3.

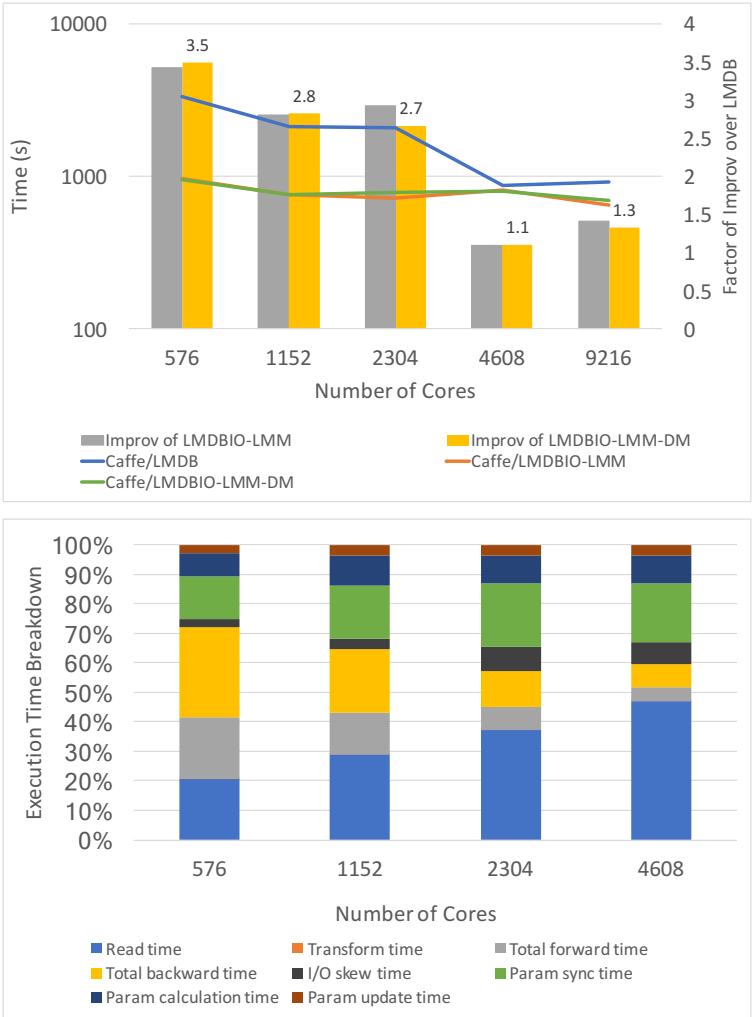

Figure 4.9: Caffe/LMDBIO-LMM-DM strong scaling on Bebop using ImageNet-Large: (a) scaling results; (b) performance breakdown

4.3.3 Evaluation of Speculative Data Reading Accuracy

The performance capability of LMDBIO-LMM-DM depends heavily on the accuracy of its estimation on what data will likely be needed for the computation in that iteration. In this section, we present a series of experiments to analyze this behavior. In our experiments, we study the accuracy of our estimation in terms of the number of pages that are needed but are not fetched during the parallel I/O phase (i.e., "missed pages") and the number of pages that are not needed but are fetched during the parallel I/O phase (i.e., "redundant pages").

In the first experiment, we measure the number of missed pages as the computation progressed through its iterations, for the CIFAR10-Large and ImageNet datasets. The experiment use 512 cores on Blues in all cases. The first two iterations result in nonzero missed pages, although for iterations after that we do not notice any missed pages for both datasets. The reason is that LMDBIO-LMM-DM automatically tunes the page range that it fetches based on the history of the accessed data in the previous iterations. That is, it corrects its estimate based on history from the prior iterations, thus allowing it to estimate the best- and worst-case bounds of access more effectively. We note that since training computations typically run for several thousands or millions of iterations, the additional missed pages during the first few iterations are mostly inconsequential for overall performance.

In our second experiment, we study the number of redundant pages read through the required iterations. Experimental results are shown in Figure 4.10. Once again, the experiment use 512 cores on Blues in all cases. We notice that the number of redundant pages increases until a certain iteration and then stabilizes. This behavior is expected because of how LMDBIO-LMM-DM works. That is since LMDBIO-LMM-DM starts with an initial estimate and then corrects this estimate based on the prior iterations, the range of pages fetched expands with iterations to cover more pages for the parallel I/O. However, once the number of redundant

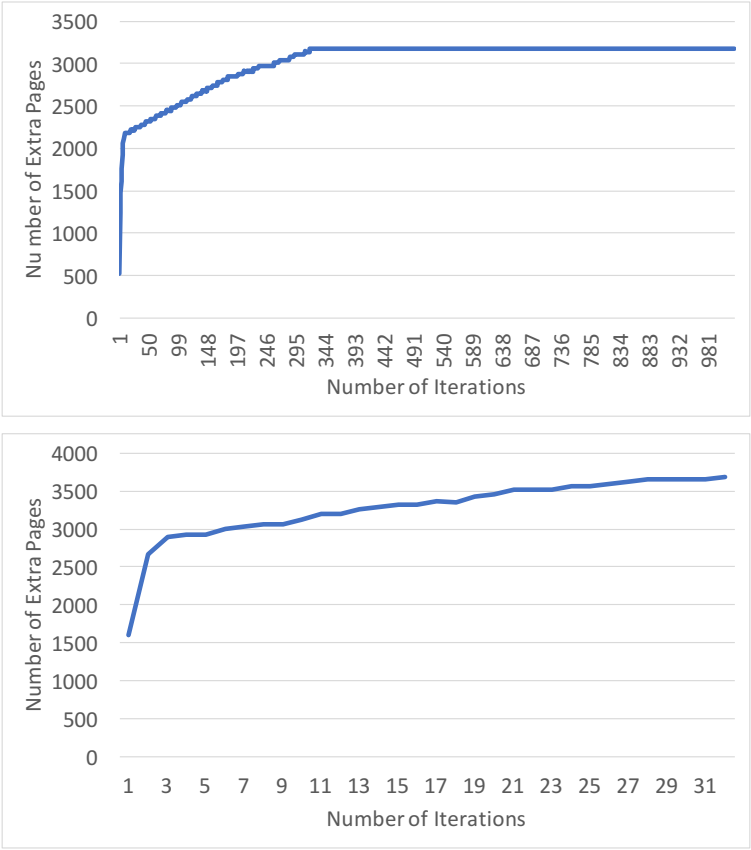

Figure 4.10: Caffe/LMDBIO-LMM-DM redundant pages read using (a) CIFAR10-Large; (b) ImageNet

pages read is large enough to not miss any page, the number of redundant pages stabilizes to a constant value.

In our third experiment, we study the missed pages with changing numbers of cores. We run the experiments for the full iteration count on Blues. We make the following observations based on Figure 4.11:

1. The number of missed pages increases with the number of cores, but the count is very small. In fact, the total number of missed pages at 512 cores is less than 700 for the CIFAR10-Large dataset, which is less than 1.4 missed pages per core, and less than 200 for the ImageNet dataset, which is less than 0.5 missed pages per core. Moreover,

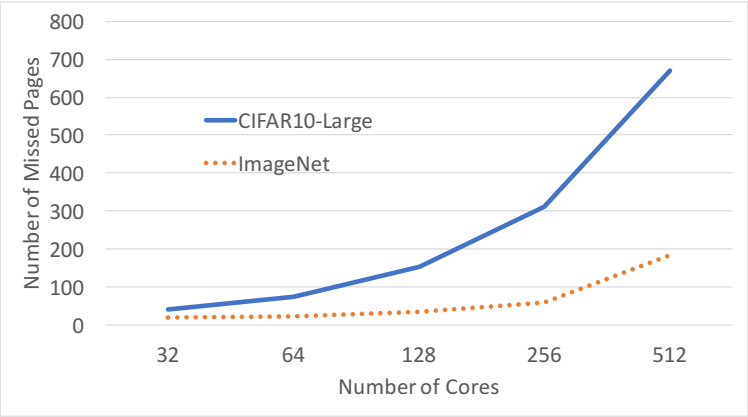

Figure 4.11: Caffe/LMDBIO-LMM-DM missed pages with varying number of cores

most of these missed pages are in the first few iterations while LMDBIO-LMM-DM is trying to converge on the range of pages to fetch.

2. The number of missed pages in the ImageNet dataset is much smaller than that in the CIFAR10-Large dataset. The reason is that the data samples are much larger in the ImageNet dataset than they are in the CIFAR10-Large dataset and, for the same amount of data processed, the ImageNet dataset covers fewer iterations than does the CIFAR10-Large dataset, thus resulting in fewer missed pages.

4.4 Chapter Summary

In this chapter, we presented LMDBIO-LMM-DM, an enhanced version of LMDBIO-LMM, our intra-node file I/O optimization presented in the previous chapter. LMDBIO-LMM-DM optimizes the I/O access of the DL framework in distributed-memory environments by coordinating between reader processes to minimize redundant file I/O. Moreover, LMDBIO-LMM-DM uses a history-based speculative data reading to enhance parallelism of its data reading. We presented the overall design and implementation of LMDBIO-LMM-DM. We also presented experimental results that show that Caffe/LMDBIO-LMM-DM can improve

the overall training time by more than 30-fold compared with the original Caffe/LMDB framework in some cases.

Chapter 5

Direct File I/O Optimizations

We have presented the analysis and optimizations of data reading on intra- and inter-node in the previous chapters (Chapters 3 and 4). In this chapter, we continue to investigate inefficiencies of LMDB.

With a thorough analysis of LMDB, we realize that the major flaw of LMDB is `mmap` as it prevents LMDB from various I/O tuning opportunities. In this section, we attempt to replace `mmap` with a more efficient file I/O—direct I/O. However, it is not trivial to remove `mmap` from LMDB since the LMDB database layout is not deterministic unless additional data is present. Here, we present a series of four direct I/O optimizations to remove `mmap` and tune the direct I/O performance.

The first two techniques incrementally replace `mmap` with direct I/O. Our first optimization, **LMDBIO-LMM-DIO**—direct I/O exploitation, manages to eliminate `mmap` almost entirely, except in the sequential seek phase that `mmap` is still required for getting the offsets and sizes of all the data records that will be read. The second optimization, **LMDBIO-LMM-DIO-PROV**—provenance information exploitation, introduces additional information to the LMDB database to enable its random database access capacity which allows us to completely remove `mmap` from our I/O path.

The latter two optimizations attempt to improve the direct I/O performance. Our third optimization, **LMDBIO-LMM-DIO-PROV-COAL**—I/O coalescing optimization, tunes the

I/O block size used to achieve the peak read bandwidth. The last optimization, **LMDBIO-LMM-DIO-PROV-COAL-STAG**—I/O staggering optimization, delays some I/O requests to address the I/O randomization problem.

In the experiments and results section, we evaluate each LMDBIO optimization using a microbenchmark and Caffe on Bebop. The results show that LMDBIO with all six optimizations outperforms LMDB in all cases.

5.1 Analysis of LMDB Inefficiencies

This section presents the analysis of LMDB inefficiencies. The problems shown in this section can be solved by using direct I/O instead of `mmap`.

5.1.1 `Mmap` Workflow Overheads

Since `mmap` performs implicit I/O, the user has no control over when an I/O operation is issued. `Mmap` needs to keep track of what data the user is trying to access. Only when the user tries to touch a piece of data can `mmap` deduce that that data segment is needed. The typical workflow used by `mmap` is as follows. When the user tries to access data that is not already available, a page fault signal is generated, which internally invokes an I/O operation. When the I/O operation completes, an interrupt is generated that marks the corresponding operation as complete. Thus, the workflow used by `mmap` is necessarily reactive based on the user data access pattern and leads to inefficiencies in the I/O path.

To showcase this inefficiency in `mmap`, we experiment to compare the I/O read bandwidth achieved by `mmap` with the bandwidth achieved by using explicit I/O (based on POSIX I/O). We develop a microbenchmark to read a 256 GB file using a single reader on a single machine.

To read the file, we use `memcpy` and `pread` with the `mmap` and POSIX I/O benchmarks, respectively. In this benchmark, we keep the read chunk size used by POSIX I/O to be 4 KB, namely, the OS page size, similar to what is used by `mmap`. In this way, the benchmark does not mix effects from the read block size with that of the `mmap` workflow. Effects of the read block size are studied separately in Section 5.1.2. The results from our experiment show that `mmap`'s read bandwidth is approximately 2.4 times lower than the read bandwidth achieved by POSIX I/O. This observation showcases the inefficiencies in `mmap`'s workflow.

5.1.2 I/O Block Size Management

As discussed earlier, most DL frameworks are iterative. In each iteration, they read one batch of data samples and process them before moving on to the next iteration. With parallel DL, this batch of data samples is further split into multiple subbatches, where each process reads a subbatch and processes it. As the number of processes increases, however, the batch of data samples is split among more processes, so each subbatch is smaller. In the extreme case, where the number of processes participating in parallel DL is equal to the number of data samples in the batch, each subbatch would contain just a single data sample. This would dramatically reduce the size of the I/O performed by each process within an iteration. For instance, with the CIFAR10-Large dataset, each data sample is just 3 KB, causing the I/O operations to be done in small page-size granularity, thus leading to significant inefficiencies.

To demonstrate the effect of I/O block size on read performance, we use the same microbenchmark discussed in Section 5.1.1, but this time we vary the read block size from 4 KB to 1 GB. Figure 5.1 shows the read I/O bandwidth of `mmap` and POSIX I/O with different I/O block sizes. Two trends are noteworthy. First, the I/O read bandwidth of POSIX I/O increases with the block size. This increase is expected and has also been demonstrated

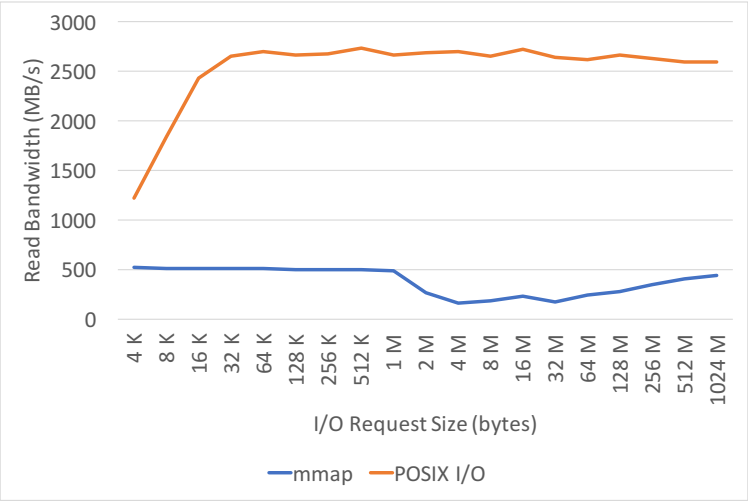

Figure 5.1: I/O block size

by other researchers in the past. Second, the block size has no impact on the I/O performance achieved by `mmap`. The reason is that `mmap` does not have information about the overall access pattern used by the application and needs to wait for the application to access data before fetching it. Even when the application uses a larger block size for performing the `memcpy` in the benchmark, this information is not passed to `mmap`. Thus, the I/O blocks used by it are inherently small. The takeaway of this analysis is that although the current I/O methodology used by LMDB cannot benefit from larger I/O blocks. If one were to migrate LMDB to using explicit I/O, larger I/O blocks could give a significant performance boost.

5.1.3 I/O Randomization

One aspect to consider while performing parallel I/O is the data access order that the various I/O requests create. For example, consider a scenario where a large number of processes need to divide a large file into smaller pieces and each process needs to access a part of it. In this example, each process issues an I/O request for its piece of the data that it needs to fetch. Since each process is independent, however, these I/O requests do not arrive at the

I/O server processes in any specific order, causing the server processes to access the file in a nondeterministic fashion. We refer to this problem as I/O randomization and illustrate it in Figure 5.2.

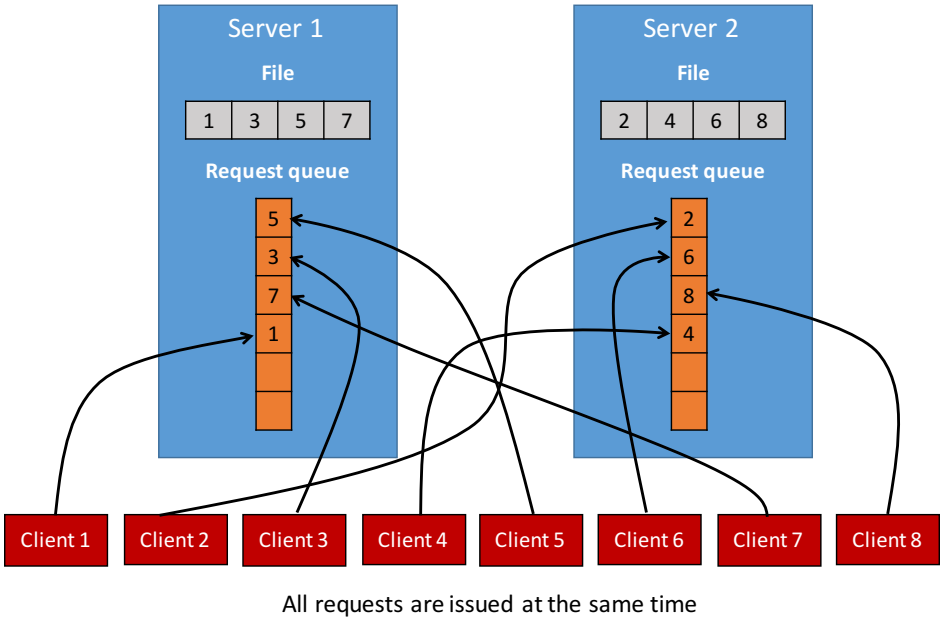

Figure 5.2: I/O randomization

I/O randomization hurts performance because, unlike sequential I/O, it cannot benefit from most I/O optimizations including data prefetching and caching, thus becoming limited by disk seek overheads. Another unfortunate aspect of I/O randomization is that as the number of processes performing parallel I/O increases, the randomization of I/O requests increases as well. Furthermore, as the read block size associated with each I/O operation increases, the impact of the additional disk seeks and the lack of benefits from data prefetching and caching increase as well. Thus, we need to carefully balance the various metrics of the amount of I/O parallelism, read block size, and I/O randomization, to maximize the overall performance.

5.2 Design and Implementation of LMDBIO-LMM-DIOs: Series of Direct I/O Optimizations

This section presents the design and implementation of four direct I/O optimizations of LMDBIO: (1) **LMDBIO-LMM-DIO** tackles `mmap`'s inefficiency; (2) **LMDBIO-LMM-DIO-PROV** addresses `mmap`'s inefficiency and sequential database access restriction of LMDB; (3) **LMDBIO-LMM-DIO-PROV-COAL** alleviates the inefficient I/O block size problem; (4) **LMDBIO-LMM-DIO-PROV-COAL-STAG** minimizes I/O randomization.

5.2.1 LMDBIO-LMM-DIO: Direct I/O Exploitation

As shown in Section 5.1.1, the implicit I/O model used by LMDB (through `mmap`) can have a significant performance impact on file I/O read. In this section, we present LMDBIO-LMM-DIO, an approach to extend LMDBIO-LMM to use direct I/O (through POSIX I/O).

The basic working model of LMDBIO-LMM-DIO is similar to that of LMDBIO-LMM. That is, LMDBIO-LMM-DIO still has a small subset of the processes designated as root processes on each node that, in turn, `mmap` the LMDB database into their respective address spaces. And, like LMDBIO-LMM, LMDBIO-LMM-DIO also creates a shared-memory buffer between all processes on the node to share the data that the root processes read from the database. The primary difference between LMDBIO-LMM and LMDBIO-LMM-DIO is that the latter uses direct POSIX I/O for performing the actual read of the data. That is, once the location of the data record in the database has been identified, LMDBIO-LMM-DIO does not use `mmap` to copy the data into the shared buffer. Instead, it computes the virtual address offset of the data record address compared with the virtual address of the start of the database and uses that offset to directly read the data using the POSIX I/O `pread` function.

We note, however, that LMDBIO-LMM-DIO does little to improve the sequential seek for locating the database records and continues to use `mmap`, just like LMDB and LMDBIO-LMM. Thus, in LMDBIO-LMM-DIO, the seek path, and the actual data read path are disjoint: the seek goes through `mmap`, whereas the actual data read goes through POSIX I/O. Because of this separation of paths, performing the seek on the same process as the one that does the actual data read is not too beneficial for LMDBIO-LMM-DIO. Therefore, we use a single process to seek through the entire database and obtain offsets and sizes for all the data records that will be used in the following training iterations. Performing the seek on a single process has the advantage of avoiding the redundant file I/O among the various root processes, although it does not help with the sequential nature of the seek. Once the seek is complete, the offsets and sizes are distributed to the other root processes, as illustrated in Figure 5.3.

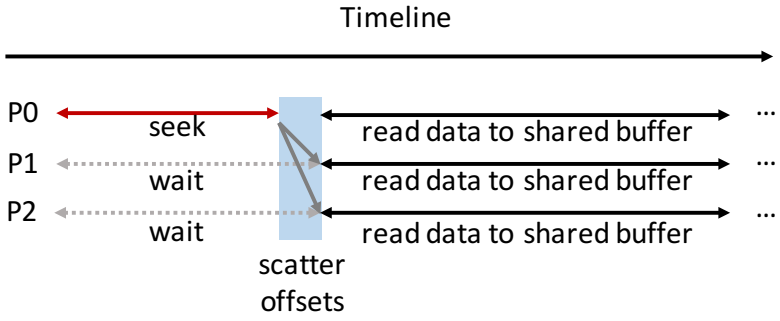

Figure 5.3: LMDBIO-LMM-DIO design: sequential seek

5.2.2 LMDBIO-LMM-DIO-PROV: Provenance Information Exploitation

LMDBIO-LMM-DM attempts to address the serialization in file I/O by performing speculative parallel reads. While that approach can be effective in reducing the redundant data

accesses in some cases, it is still an approximation technique and can cause a significant increase in the file I/O if the approximation is incorrect. Unfortunately, no way exists to precisely estimate the location of the data records without the sequential seek or additional information. The reason is that the layout of the LMDB database depends not only on the content of the database but also on the way the database was created. This information is not natively stored in the LMDB database file.

In this section, we propose LMDBIO-LMM-DIO-PROV, a technique that provides a more elegant alternative to address the serialization in file I/O, compared with LMDBIO-LMM-DM, by completely and deterministically eliminating the sequential seek restriction of LMDB. The catch, however, is that LMDBIO-LMM-DIO-PROV requires the user to provide more information than what the LMDB database natively provides. We refer to this information as the "database provenance information."

LMDB Database Creation

Before explaining the provenance information that we require for LMDBIO-LMM-DIO-PROV, we briefly summarize how the LMDB database creation process works. LMDB employs a multi-version concurrency control policy to guarantee data integrity and reliability in the multi-reader & single-writer model. This model allows a reader to read a valid snapshot of the database without acquiring a lock. Locking is required only when writing to the database. To provide concurrency, LMDB adopts a "copy-on-write" policy on the database file where new data is written to the file without overwriting or relocating old data. Any change to existing data in the database file, however, will be applied to a copy of that data. In other words, LMDB will copy existing data to a new location and apply changes to the new resource when a write occurs. This policy ensures that data in the file is always in a valid state.

Since LMDB is a transactional database, it operates at the granularity of transactions. When new data is added to the database, it will be written to permanent storage only when that transaction is committed. During the *commit*, the layout of the database file is modified. Resources that have been modified will be duplicated. For LMDB, these modifiable resources are the branch pages and leaf pages. When the tree structure changes, some of existing branch and leaf pages are modified to update their connectivity to other pages (i.e., neighboring and children pages). With LMDB the tree grows in a bottom-up manner where pages that contain data (i.e., key-value pairs) are added first. Each leaf/branch page has a limit on the number of children that it can have. New pages are added to the tree when the number of children in that page has reached that limit.

An example of LMDB database creation is demonstrated in Figure 5.4. $State1$ shows an example initial state of the database where all prior transactions have been successfully committed (i.e., the previous data is in the disk and is identical to the content in memory $1A$). In $State2$, data page $O4$ is appended to the tree causing the leaf page $L1$ to be modified. In this case, LMDB copies $L1$ to a new location before modifying it (memory $2A$). Then, the old memory location of $L1$ is marked as free (memory $2B$). After that, $O4$ will be added to the database file. In this example, we assume that $O4$ can fit in the free memory region (memory $2C$). Otherwise, it will be appended to the end of the memory area. Suppose that the transaction has not yet been committed. $State3$ shows how the tree grows in the case that the number of children of $L1$ exceeds its limit (i.e., 4 children). In that case, a new leaf page ($L2$) and a new branch page ($B1$) are added to the tree.

LMDB Provenance Information

As explained earlier, the location of data records in the LMDB database cannot be determined by using only the natively available information in the database metadata. Fortu-

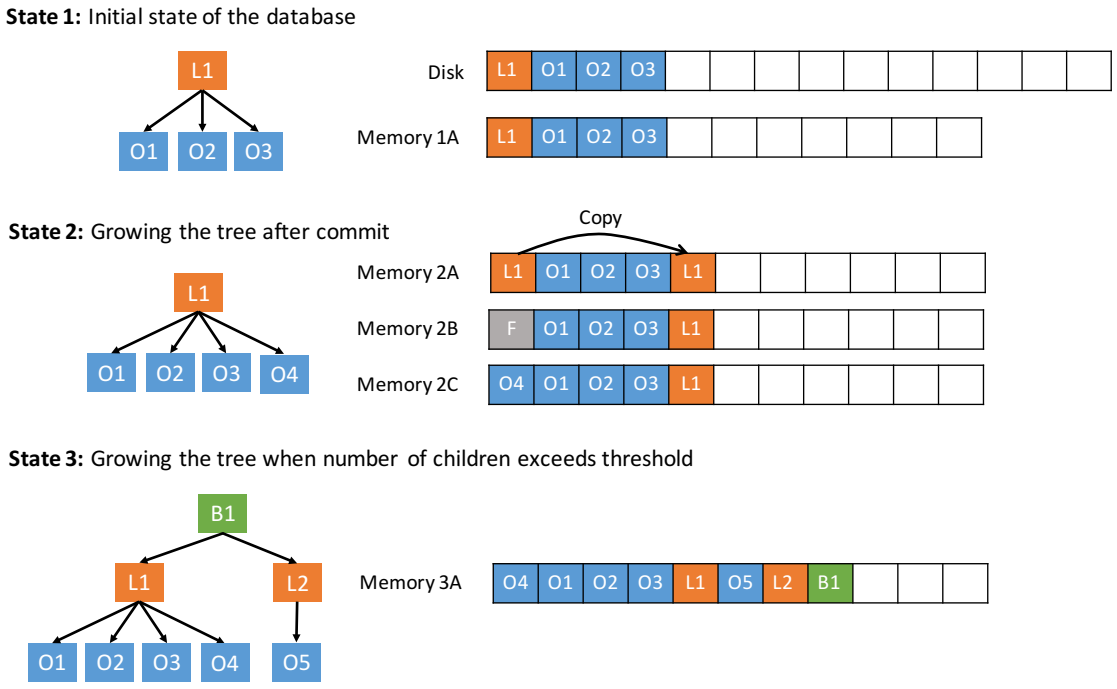

Figure 5.4: LMDB database creation example

nately, LMDB uses a deterministic algorithm to create the B+ tree database. Thus, with additional information about the database creation (i.e., the database provenance information), we can dynamically compute the database layout. This computation allows us to precisely deduce the accurate location of each database record, completely eliminating the seek.

In LMDBIO-LMM-DIO-PROV, we propose maintaining a separate auxiliary file for each LMDB database file that contains the following provenance information: (1) frequency that the transactions are committed in, (2) maximum number of records that a leaf node can contain, (3) maximum number of children that a branch node can have, (4) size of each data record, (5) order in which the data records are added, and (6) number of LMDB metadata pages. This provenance information can be collected either when the database is being generated or later as one-time postprocessing of the database. We note that the

proposed provenance information is typically small compared with the database itself (i.e., a few hundred bytes).

Once such provenance information is available, its usage in LMDBIO-LMM-DIO-PROV is straightforward. Each root process computes the offsets of all the data records that it needs by following the algorithm that is adopted by LMDB for creating the database. This computation adds negligible cost compared with the cost of the I/O itself. Once the offsets are calculated, the actual file I/O is done through POSIX I/O, similar to LMDBIO-LMM-DIO. We note that without the additional provenance information LMDBIO-LMM-DIO-PROV cannot be used and we would need to fall back to LMDBIO-LMM-DM for improving the sequential seek.

An important aspect to note here is that any improvement to the seek time needs to be taken with a grain of salt. For example, in cases where the application iterates over the data for a very large number of epochs, one might be able to simply store the database offsets in memory to be used in later epochs. However, such an approach raises a few concerns that must be kept in mind.

1. It is practical only if the number of data samples is an exact multiple of the number of processes. Any offset in this would mean that the data samples computed by a given process would not be exactly the same in every epoch. In cases where the number of data samples is not an exact multiple of the number of processes, one can divide the data samples as evenly as possible across the different processes and then treat the remainder separately. While this might seem like an enticing possibility, however, we note that it would change the semantics of the LMDB model. With its current semantics, the database is treated as a circular collection of records, so one would return to the first record after the last record has been read. This allows applications

using LMDB to be guaranteed that the read of a block of records always returns the full block of records and never a partial block. If we treat the remainder separately, those semantics would no longer be true. As a consequence, such a change in the semantics would, in turn, require intrusive modification to the entire LMDB ecosystem.

2. The efficiency of this approach depends on how many epochs of training are used. For cases where the database is extremely large, some algorithms tend to rely on a single-pass analysis (i.e., the database is read only once) or on analyzing the data using just a few epochs. In such cases, the seek overhead can still be significant, and the provenance information that we proposed in this section can help.

3. While data could theoretically be streamed from an online source, such a model is not as common today. Training datasets are typically stored in persistent files and used for training with multiple models or multiple parameter settings.

4. Similarly, while splitting the dataset into a large number of files is possible (e.g., one file per process), so as to completely avoid seeking, such practice is strongly discouraged on most large supercomputing systems. The reason is that reading from a large number of files can easily overwhelm the metadata server, causing the filesystem to suffer from significant performance loss or even crash [110].

5.2.3 LMDBIO-LMM-DIO-PROV-COAL: I/O Coalescing Optimization

As mentioned in Section 5.1.2, as the parallelism used by the DL algorithm increases, the size of the subbatch used by each process decreases. In the extreme case, when the parallelism used for the DL training is as large as the number of available data samples in each batch,

each process would need a single data sample in each iteration. Thus, each root process would end up reading smaller blocks of data. As an example, if we consider the CIFAR10-Large database, when using 9,216 processes with a batch size of 18,432, each process would need just two data samples in every iteration, where each data sample would be 4 KB in size (3 KB actual data). Even if we use a single root process on each node, the root process would perform an I/O of 288 KB in every iteration. Most filesystems, however, require much larger block sizes (typically in multiple megabytes) for optimal I/O performance.

We tackle this issue in LMDBIO-LMM-DIO-PROV-COAL by allowing it to assume the iterative nature of DL applications. That is, even though a single iteration does not require too much data, if we can coalesce the data required in multiple iterations, we can increase the block size used in each I/O operation. With LMDBIO-LMM-DIO-PROV-COAL, each root process reads a large contiguous chunk of data (large enough to saturate the I/O performance of the filesystem). LMDBIO-LMM-DIO-PROV-COAL tunes the I/O block size that it uses so as to limit the amount of memory that it consumes for I/O (kept at 2.5 GB in our experiments). Thus, as the parallelism in the DL training increases, it fetches data required for more iterations within a single I/O operation.

5.2.4 LMDBIO-LMM-DIO-PROV-COAL-STAG: I/O Staggering Optimization

Our last optimization technique, LMDBIO-LMM-DIO-PROV-COAL-STAG, addresses the I/O randomization problem presented in Section 5.1.3. The general idea used by LMDBIO-LMM-DIO-PROV-COAL-STAG is to limit the number of I/O operations that are issued simultaneously so as to minimize such randomization while maintaining sufficient parallelism to maximize I/O performance. To achieve this goal, we use a technique called I/O staggering.

In this technique, the root processes are divided into multiple groups of the same size. Root processes that access segments of the file that are close to each other are grouped together. Once the grouping is done, we allow one group of root processes (referred to as a staggering group) to access the file concurrently while the remaining groups wait for the previous groups to complete their I/O. We use a token-passing approach: a process can perform I/O only when it has a token. Suppose the staggering group size is n. Then there are n tokens, with the root processes in each group labeled from 0 to $n-1$. When a root process is done with its I/O, it passes on its token to the root process in the next group with the same label as itself. We simply use MPI send/recv to pass tokens between processes.

We note that the staggering size needs to be carefully selected. A substantial staggering size would lead to increased randomization, while a very small staggering size would lead to reduced parallelism in I/O. We empirically evaluated the best staggering sizes for a different number of processes and used them for our experiments.

We also note that more elegant approaches for managing I/O staggering exist than those we propose in this work. One example would be to use the POSIX file-locking mechanism. That is, each group would attempt to lock the database file; and once it acquired the lock, it would perform the actual I/O. This approach would achieve the same outcome as our token-passing approach and would further remove the unnecessary and artificial ordering restriction that the proposed token-passing approach forces. Unfortunately, most distributed filesystems (e.g., Network File System or NFS) do not provide strict POSIX semantics, including `fcntl` and file locking [158], thus making its portability questionable. Therefore, we used the proposed token-passing approach as a workaround to this particular shortcoming of some filesystems.

5.3 Direct File I/O Optimization Experiments and Results

In this section, we compare the performance of LMDBIO with that of LMDB. In Section 5.3.1 we evaluate the performance of each of the proposed optimizations using simple microbenchmarks. The purpose of this evaluation is to understand the benefits and shortcomings of each optimization without diluting the results with other computation that would happen in a typical DL application. In Sections 5.3.2 and 5.3.3, we use strong- and weak-scaling experiments to compare the performance of Caffe/LMDBIO with that of the original Caffe/LMDB. The purpose of this evaluation is to understand the impact of LMDBIO on the overall performance of the Caffe DL framework on real datasets. Our experiments use the datasets, networks, and supercomputer systems described in Chapter 3.1.1.

5.3.1 Microbenchmark Evaluation and Analysis

This section consists of evaluations of our LMDBIO direct I/O optimizations using the same microbenchmark that is presented in Chapter 3.4.1. The benchmark contains only the file I/O portion of Caffe, but not the training part. The experiment is conducted on Bebop using the CIFAR10-Large dataset with a batch size of 18,432 images (9.4 million images).

LMDBIO-LMM-DIO

We compare the read performance of LMDBIO-LMM-DIO with that of LMDB, LMDBIO-LMM, and LMDBIO-LMM-DM, as shown in Figure 5.5(a). LMDBIO-LMM-DIO achieves better performance than the other approaches in almost all cases primarily because of its usage of POSIX I/O for data reading in place of `mmap`. In some cases, however, LMDBIO-

LMM-DM slightly outperforms LMDBIO-LMM-DIO. The reason is that LMDBIO-LMM-DIO does nothing to optimize the seek, a process that can take a significant amount of time. In fact, as shown in our read time breakdown in Figure 5.5(b), the seek in LMDBIO-LMM-DIO takes up nearly 20% of the read time. Nevertheless, LMDBIO-LMM-DIO still outperforms LMDB by 17.18-fold on 4,608 cores.

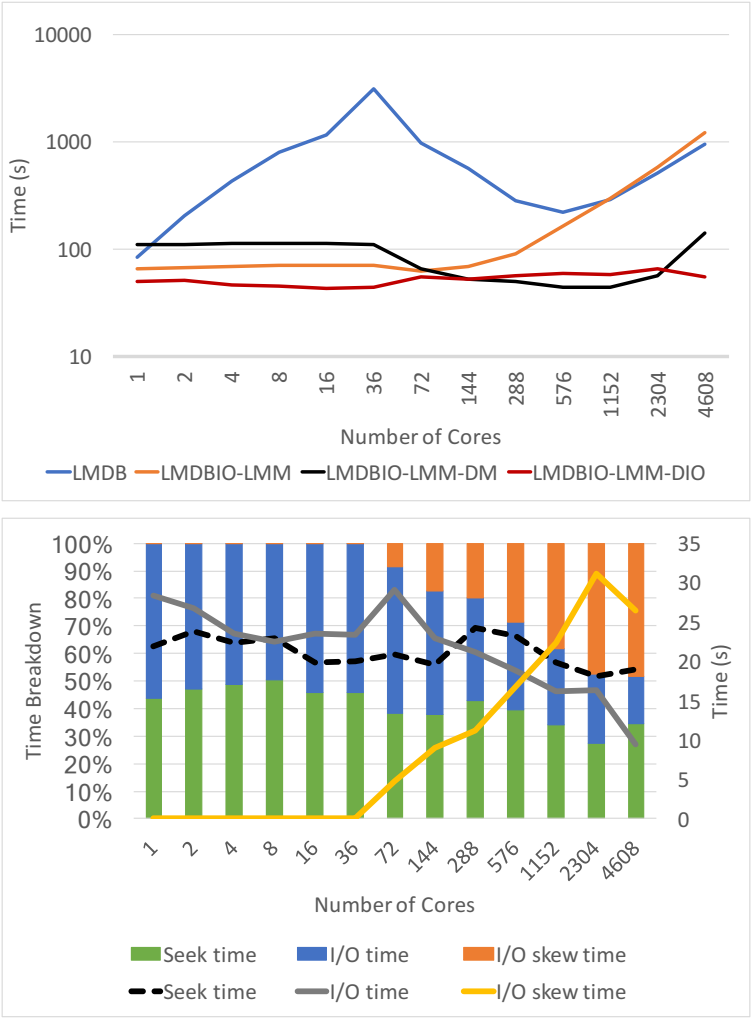

Figure 5.5: LMDBIO-LMM-DIO performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, and LMDBIO-LMM-DM; (b) total read time breakdown

We note that LMDBIO-LMM-DIO still suffers from data skew, similar to LMDBIO-LMM and LMDBIO-LMM-DM. Unlike LMDBIO-LMM, however, this skew is not because of data

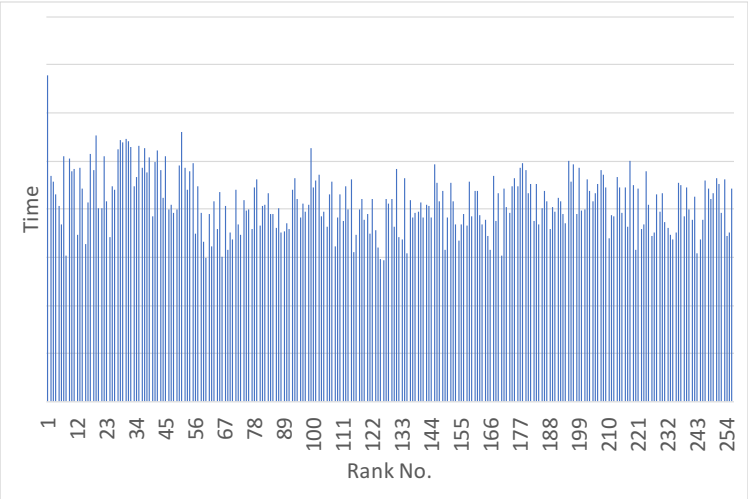

Figure 5.6: LMDBIO-LMM-DIO I/O skew analysis

prefetching, which we verified by measuring the I/O time on each reader rank as shown in Figure 5.6. Instead, the skew is due to other serialization in the file I/O such as that related to I/O randomization.

LMDBIO-LMM-DIO-PROV

Figure 5.7(a) compares the performance of LMDBIO-LMM-DIO-PROV with that of LMDB, LMDBIO-LMM, LMDBIO-LMM-DM, and LMDBIO-LMM-DIO. LMDBIO-LMM-DIO-PROV consistently outperforms all the existing approaches, achieving 19.44-fold improvement in performance on 4,608 cores compared with LMDB. The performance improvement in LMDBIO-LMM-DIO-PROV is attributed to its elimination of the sequential seek to access the database records. This improvement in performance highlights the importance of the database provenance information in scalable DL.

Despite the impressive gains in performance, however, LMDBIO-LMM-DIO-PROV still suffers from some shortcomings that cause its I/O time to increase as the number of cores increases. We plotted this behavior in Figure 5.7(b). This figure shows that a significant

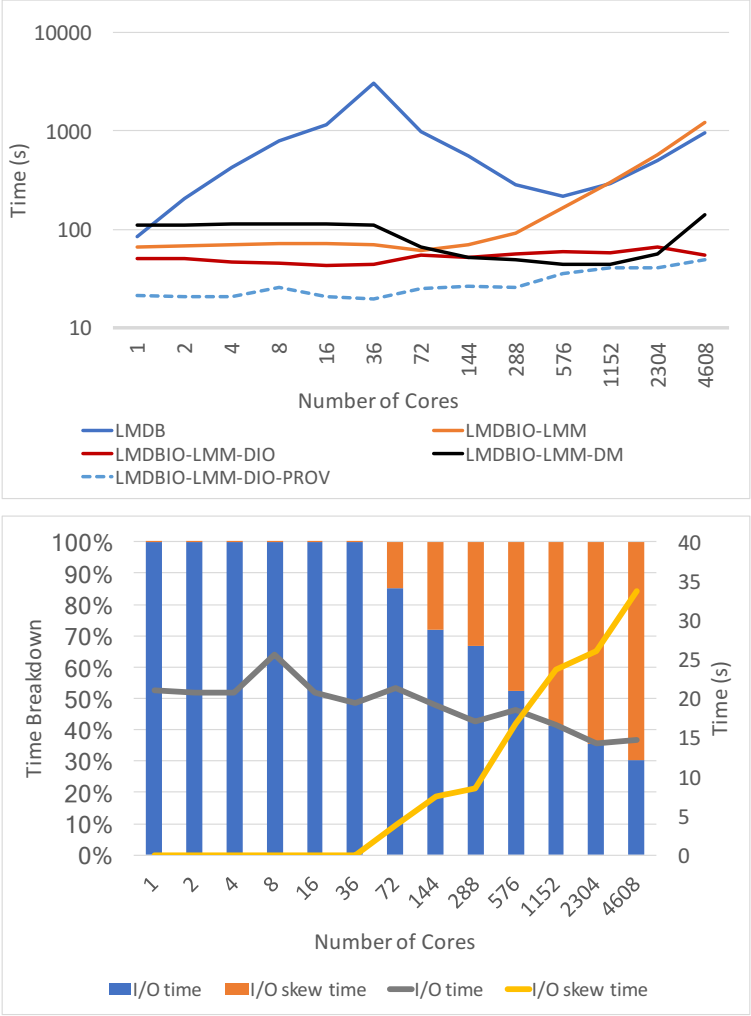

Figure 5.7: LMDBIO-LMM-DIO-PROV performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, and LMDBIO-LMM-DM; (b) total read time breakdown

portion of the I/O time is taken by the skew between the different processes, which is an artifact of the I/O randomization described in Section 5.1.3.

LMDBIO-LMM-DIO-PROV-COAL

Figure 5.8(a) compares the performance of LMDBIO-LMM-DIO-PROV-COAL with that of LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, and LMDBIO-LMM-DIO-PROV and demonstrates that LMDBIO-LMM-DIO-PROV-COAL consistently achieves the best performance among all approaches. In fact, LMDBIO-LMM-DIO-PROV-COAL outperforms LMDB by 21.86-fold on 4,608 cores. The primary performance gain in LMDBIO-LMM-DIO-PROV-COAL comes from the fact that it optimizes the I/O block size by coalescing data required in multiple iterations into fewer I/O operations. This approach better utilizes the I/O subsystem, resulting in improved performance.

A breakdown of the read time in Figure 5.8(b) shows that LMDBIO-LMM-DIO-PROV-COAL reduces the skew time to around 25% of the total I/O time. While the actual read operation now takes most of the time, room for improvement still remains.

LMDBIO-LMM-DIO-PROV-COAL-STAG

Figure 5.9(a) compares the performance of LMDBIO-LMM-DIO-PROV-COAL-STAG with that of LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DIO-PROV, and LMDBIO-LMM-DIO-PROV-COAL. The figure shows that LMDBIO-LMM-DIO-PROV-COAL-STAG performs the same as or better than all the other techniques, outperforming LMDB by 81.05-fold on 4,608 cores. This improvement in performance is attributed to the reduced I/O randomization in LMDBIO-LMM-DIO-PROV-COAL-STAG.

Our analysis of the I/O time breakdown is shown in Figure 5.9(b). This figure, however,

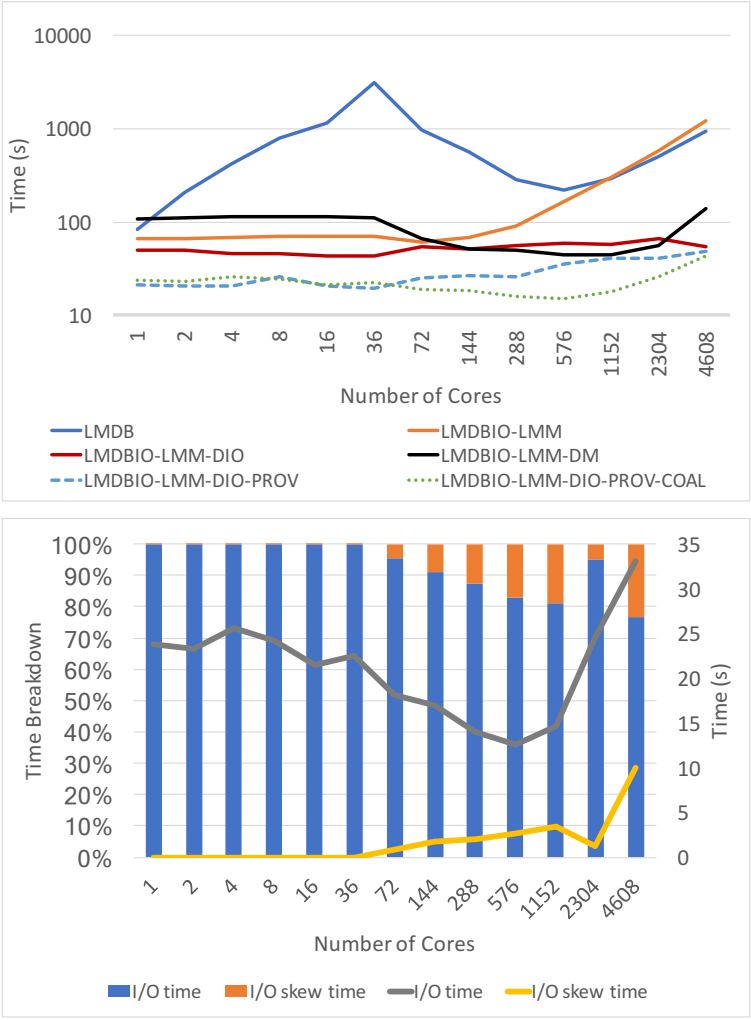

Figure 5.8: LMDBIO-LMM-DIO-PROV-COAL performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DM, and LMDBIO-LMM-DIO-PROV; (b) total read time breakdown

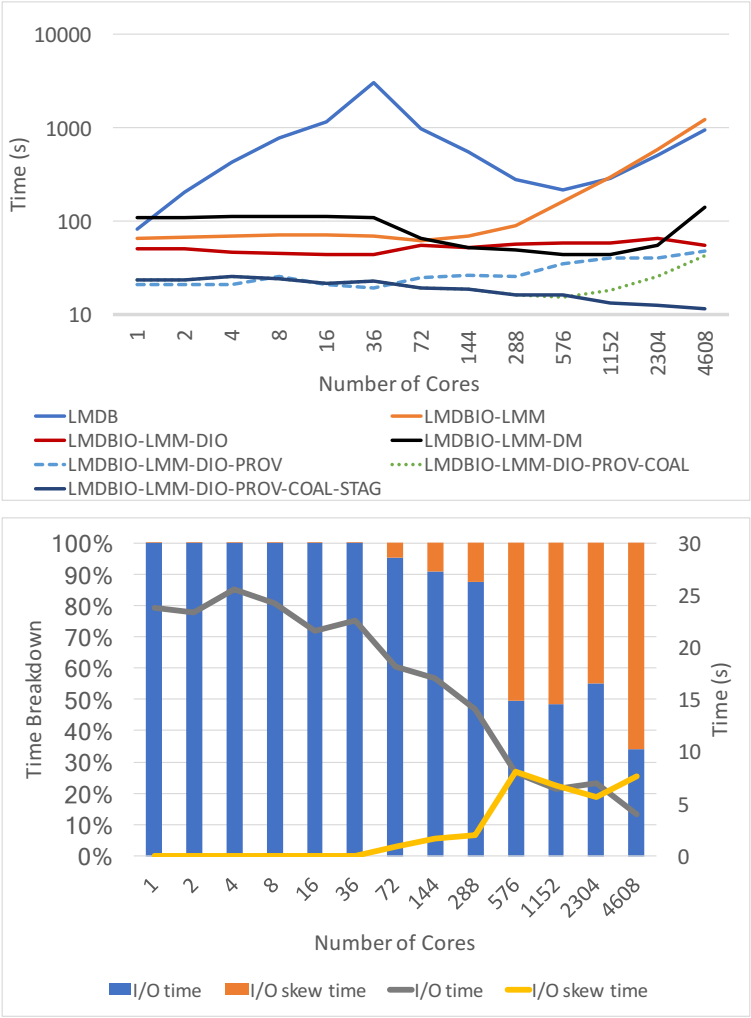

Figure 5.9: LMDBIO-LMM-DIO-PROV-COAL-STAG performance analysis: (a) read performance compared with LMDB, LMDBIO-LMM, LMDBIO-LMM-DIO, LMDBIO-LMM-DM, LMDBIO-LMM-DIO-PROV, and LMDBIO-LMM-DIO-PROV-COAL; (b) total read time breakdown

can be a bit misleading. While it shows a significant increase in I/O skew compared with LMDBIO-LMM-DIO-PROV-COAL, this skew is intentional. That is, because LMDBIO-LMM-DIO-PROV-COAL-STAG groups the root processes and forces only one group to be actively performing I/O at a given point in time, it artificially appears that there is high I/O skew time. Nevertheless, LMDBIO-LMM-DIO-PROV-COAL-STAG comprehensively outperforms all the other presented techniques.

5.3.2 Strong-Scaling Evaluation of Caffe Deep Learning Training

This section presents the evaluation results of our LMDBIO direct I/O optimizations using the actual DL benchmark, Caffe. The experiments are run on Bebop.

As described in Chapter 3.1.1, all of our experiments so far use single-threaded MKL while achieving parallelism on the node using multiple processes. An alternate approach that one might consider is to use a single process on each node but to take advantage of intra-node parallelism through the multithreaded Intel MKL library, so as to utilize the cores better. While at first blush that seems promising, such an approach would, by definition, only utilize the cores on the node during MKL operations, while the rest of the computational workflow would remain sequential, thus wasting cores. We have included the multithreaded MKL version (denoted LMDB-MT-MKL) in the experiments in this section for completeness, despite its known inefficiency especially when the number of cores is large.

Figure 5.10 presents the strong-scaling results for CIFAR10-Large using a batch size of 18,432 and the training iterations of 512. Figure 5.10(a) shows the execution time of Caffe with the different frameworks, and Figure 5.10(b) shows the factor of improvements compared with Caffe/LMDB. All Caffe/LMDBIO optimizations outperform Caffe/LMDB in all cases, with Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG achieving nearly 65-fold performance

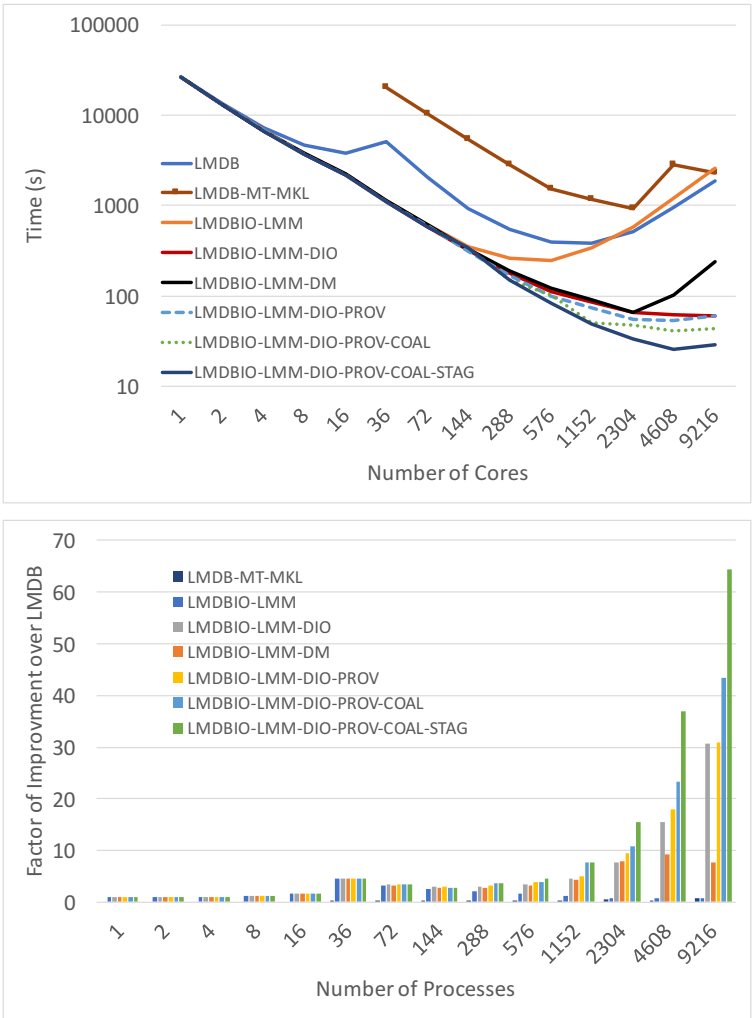

Figure 5.10: Strong scaling using CIFAR10-Large on Bebop: (a) total execution time; (b) factor of improvement over Caffe/LMDB

improvement over Caffe/LMDB on 9,216 cores.

Figure 5.11 shows execution time breakdown of each optimization. Note that graph scales are not the same as each optimization has a different absolute total execution time. The execution time breakdown of each optimization is similar to the one that we obtain in the microbenchmark evaluation.

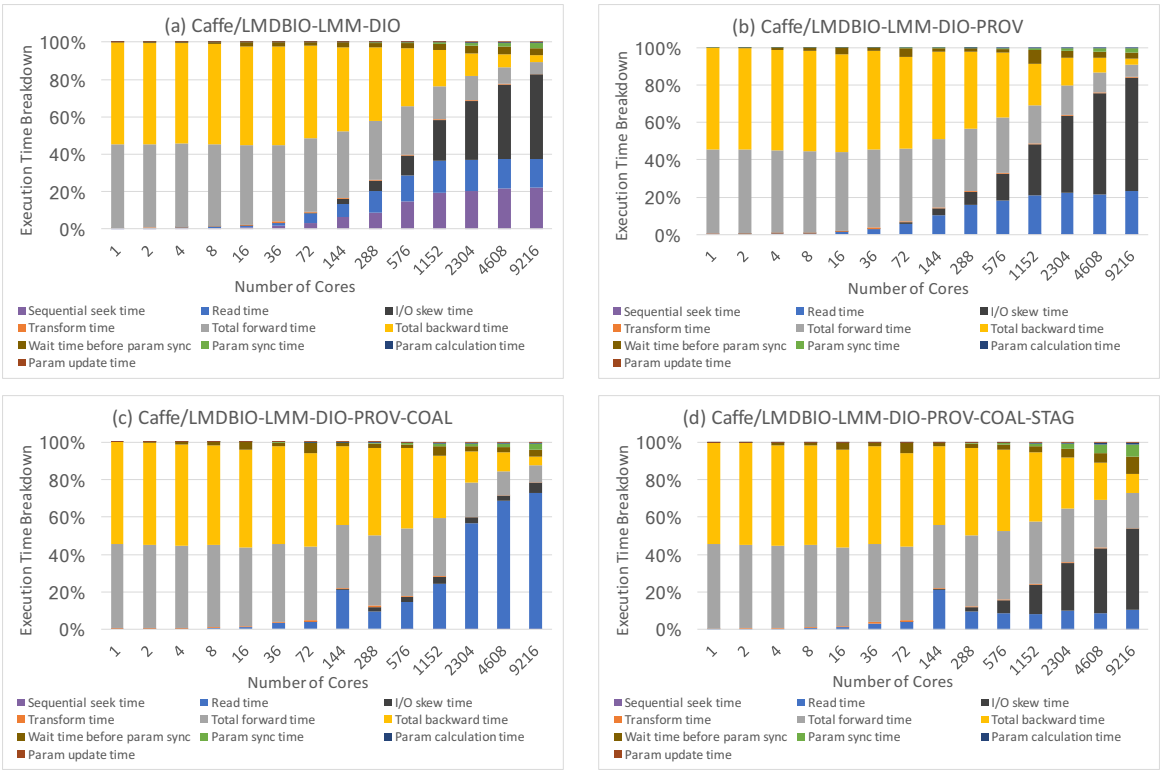

Figure 5.11: Execution time breakdown using CIFAR10-Large on Bebop: (a) Caffe/LMDBIO-LMM-DIO; (b) Caffe/LMDBIO-LMM-DIO-PROV; (c) Caffe/LMDBIO-LMM-DIO-PROV-COAL; (d) Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG

Figure 5.12 shows strong-scaling results for ImageNet-Large using a batch size of 18,432 and the training iterations of 32. Figure 5.12(a) shows the execution time of Caffe with the different frameworks, and Figure 5.12(b) shows the factor of improvements compared with Caffe/LMDB. The general performance trend observed in the figures is similar to that with CIFAR10-Large, although the performance improvements are smaller. The reason is that

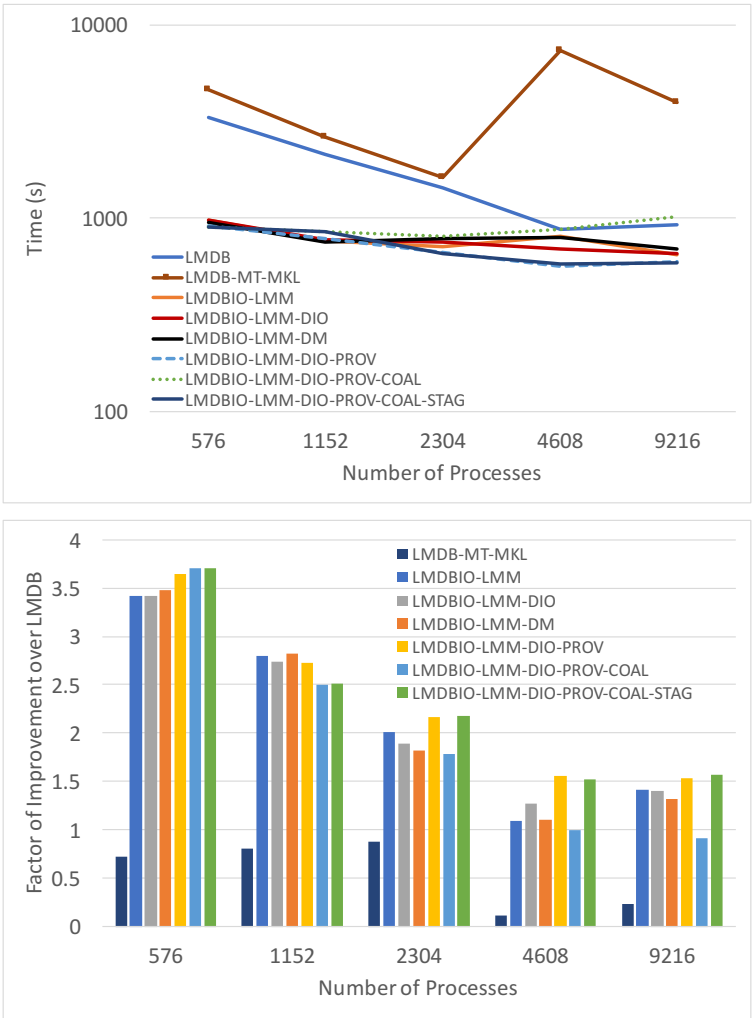

Figure 5.12: Strong scaling using ImageNet-Large on Bebop (a) total execution time; (b) factor of improvement over Caffe/LMDB

the structures of the two datasets are different. Specifically, ImageNet-Large contains larger data sample sizes (192 KB for ImageNet-Large compared with 3 KB for CIFAR10-Large), resulting in significantly different I/O characteristics. For example, header access is a small fraction of I/O for ImageNet-Large, whereas it is a significant portion of I/O for CIFAR10-Large; in other words, the header and the data are on the same physical page in memory for CIFAR10-Large, so accessing one without the other is difficult. Another example is that of I/O randomization, which has a significantly higher impact on ImageNet-Large than it does on CIFAR10-Large because of the larger sizes of the data samples, making each batch of samples typically larger than the I/O request size of the filesystem.

An interesting trend that we observe is that for the ImageNet-Large dataset, Caffe/LMDBIO-LMM-DIO-PROV-COAL performs worse than other techniques, particularly when the number of cores is large. The reason is that although all techniques other than Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG suffer from I/O randomization, Caffe/LMDBIO-LMM-DIO-PROV-COAL is particularly susceptible because this technique actively increases the amount of data that each process reads through coalescing. Thus, in Caffe/LMDBIO-LMM-DIO-PROV-COAL, if I/O requests arrive out of order at the I/O server, the data segments accessed by these requests are especially far away for ImageNet-Large because of the large size of its data samples, thus causing further degradation in performance. As expected, once I/O staggering is applied in Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG, this performance degradation goes away. In fact, Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG outperforms all other approaches, gaining approximately 1.6-fold performance over Caffe/LMDB on 9,216 cores.

Execution time breakdowns are demonstrated in Figure 5.13. The breakdown of each optimization is similar to those of the microbenchmark and CIFAR10-Large except that the ImageNet-Large training has a bigger portion of communication time (denoted as “Param

sync time"). This demonstrates that once we minimize the file I/O bottleneck, the effect of the network I/O has become more significant.

Figure 5.13: Execution time breakdown using ImageNet-Large on Bebop: (a) Caffe/LMDBIO-LMM-DIO; (b) Caffe/LMDBIO-LMM-DIO-PROV; (c) Caffe/LMDBIO-LMM-DIO-PROV-COAL; (d) Caffe/LMDBIO-LMM-DIO-PROV-COAL-STAG

The performance in term of a number of trained images per second is shown in Figure 5.14. In the case that a number of images and a number of cores are given, the information shown in the graph can be used to estimate the amount of time that Caffe with each I/O framework uses to complete the training. For instance, in the best case, LMDBIO-LMM-DIO-PROV-COAL-STAG can finish one epoch of CIFAR10-Large (50 million samples) in 2 hours and 16 minutes on 4,608 cores (364,788 images per second), while it can complete an epoch of ImageNet-Large (6 million images) in 24 hours (4,100 images per second). However, we note that our datasets are not conventional as they are amplified from the original datasets. Therefore, the results here cannot be compared with the ones shown in the existing literature.

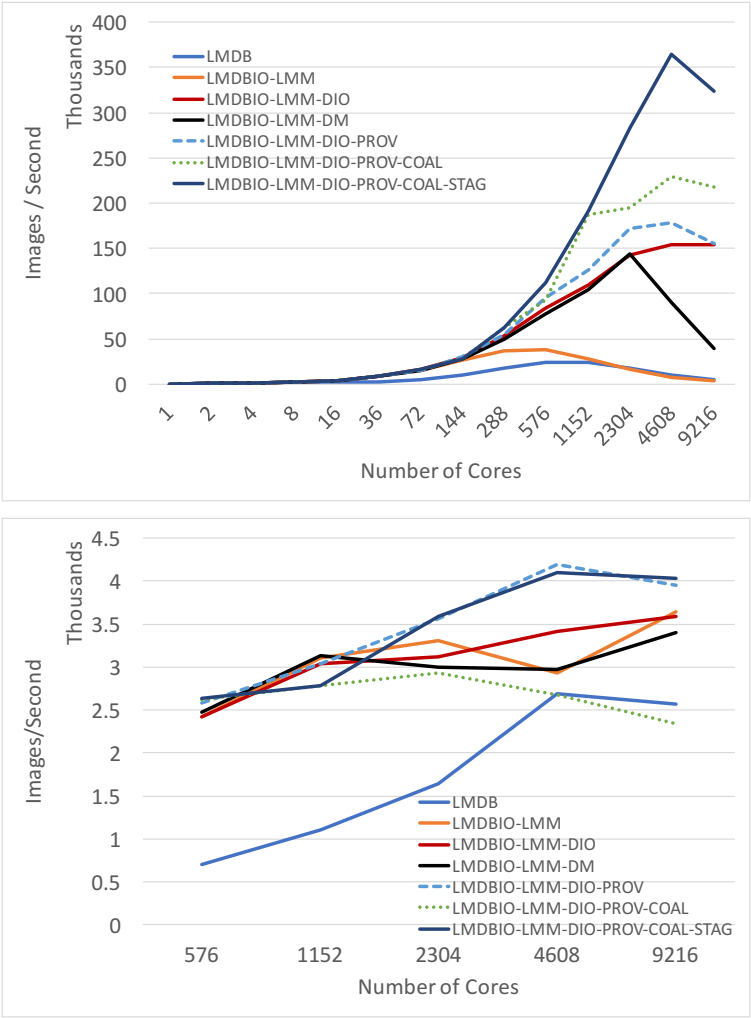

Figure 5.14: Images per second of Caffe/LMDBIO using (a) CIFAR10-Large; (b) ImageNet-Large

5.3.3 Weak-Scaling Performance Evaluation of Caffe Deep Learning Training

Apart from the strong-scaling experiments shown so far, we also conduct a weak-scaling evaluation of LMDBIO on Bebop. Here, we increase the total batch size (i.e., the total number of images processed by all processes together in each iteration) by k times when the process count is increased by k times. The subbatch size (i.e., the number of samples that a single process computes in each iteration) is set to two. We choose to keep the total number of processed data samples constant throughout the weak-scaling experiments to 9,437,184 and 2,359,296 samples for CIFAR10-Large and ImageNet-Large, respectively. Thus, when the number of processes doubles, the total batch size doubles, and the number of iterations halves. We note that because each iteration is bulk synchronous, increasing the number of iterations would not change the performance trend showcased in the graphs—all the performance numbers for a given number of processes would simply be multiplied by a constant factor.

The weak-scaling results for CIFAR10-Large are illustrated in Figure 5.15 and those of ImageNet-Large are illustrated in Figure 5.16. We observe trends for our weak-scaling experiments similar to those for the strong-scaling experiments. For weak scaling, Caffe/LMDBIO outperforms Caffe/LMDB by up to 43 times for CIFAR10-Large and by up to 1.9 times for ImageNet-Large. In the case of ImageNet-Large, LMDBIO-LMM-DIO-PROV-COAL-STAG achieves the same performance as LMDBIO-LMM-DM. This is expected. LMDBIO-LMM-DM is an effective approach in improving performance—the drawback of LMDBIO-LMM-DM is not that it cannot improve performance but that the approach itself is speculative. That is, in some cases, the speculation might work well while in other cases the speculation might result in additional I/O causing some performance loss. The direct I/O methods (all

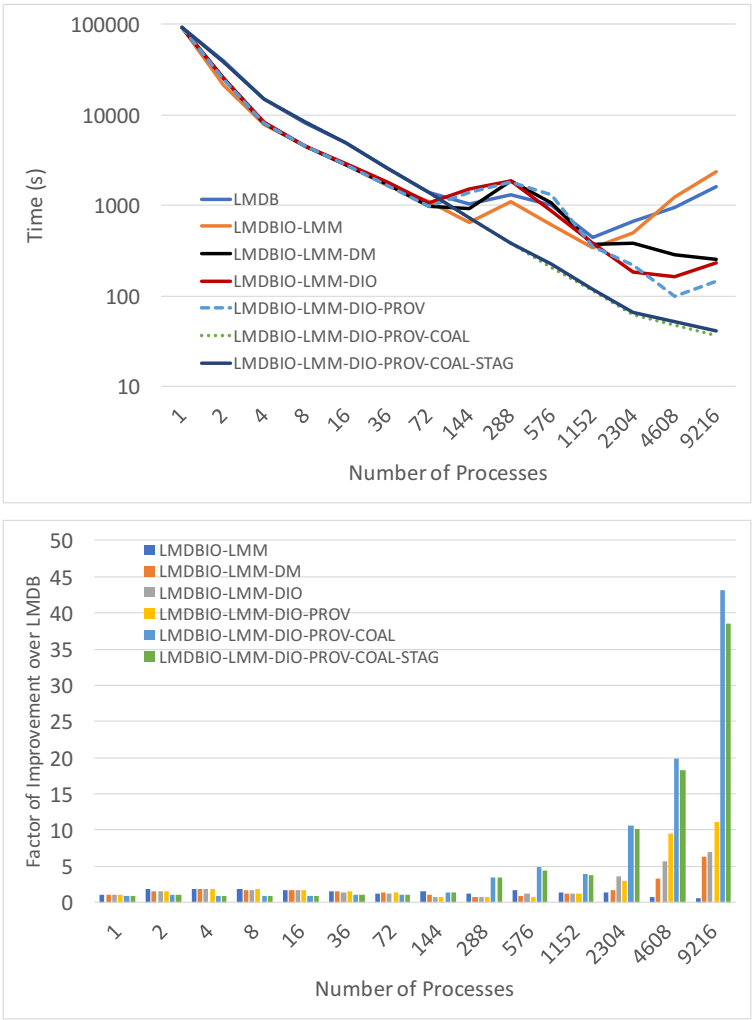

Figure 5.15: Weak scaling using CIFAR10-Large on Bebop: (a) total execution time; (b) factor of improvement over Caffe/LMDB.

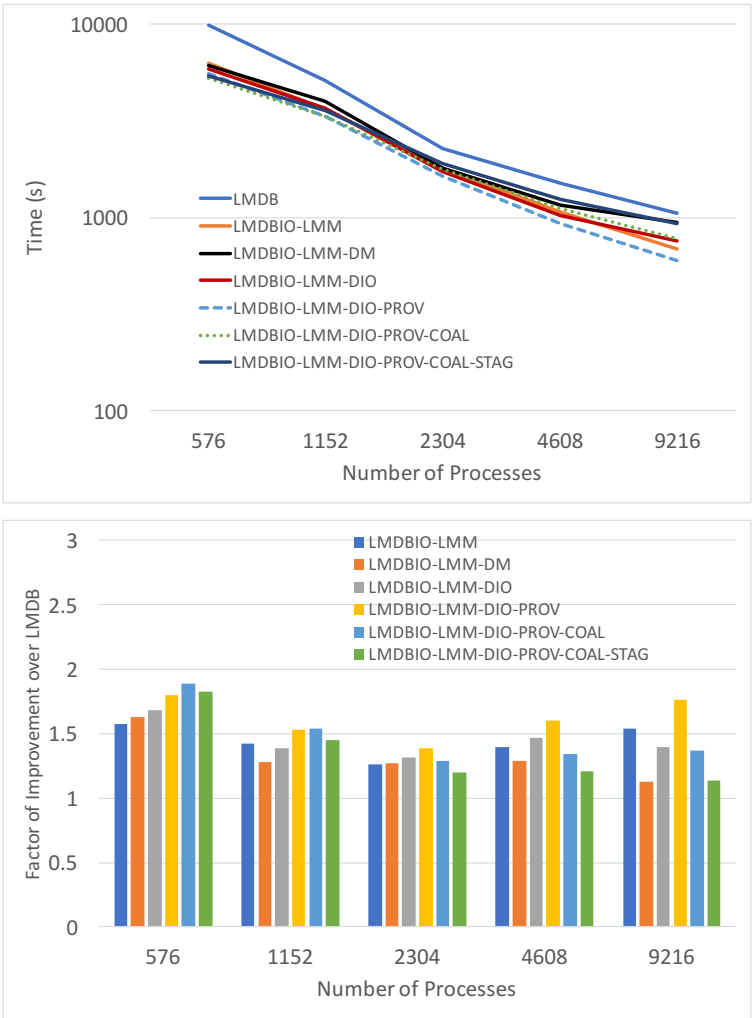

Figure 5.16: Weak scaling using ImageNet-Large on Bebop: (a) total execution time; (b) factor of improvement over Caffe/LMDB.

optimizations with the LMDBIO-LMM-DIO prefix), on the other hand, deterministically improve performance without using such speculation. Thus, they are better approaches in the general case.

5.4 Chapter Summary

In this chapter, we presented our continuous ambition to improve data I/O performance of large-scale DL frameworks. We continued to analyze state-of-the-art I/O subsystem of DL, LMDB. We have identified that the core file I/O engine of LMDB, namely `mmap`, was the largest source of data reading inefficiencies. Therefore, we proposed LMDBIO direct I/O optimizations that leverage POSIX I/O for data reading instead of `mmap`. Our final direct I/O optimization, LMDBIO-LMM-DIO-PROV-COAL-STAG, is able to maximize the available I/O performance on our experimental system.

Chapter 6

Computational Imbalance Optimizations for Data Processing

In the previous chapters, we have addressed one aspect of the scalability bottlenecks, that is the file I/O problem. In this chapter, we investigate another important aspect of DL scalability that is usually overlooked, *computational imbalance*, which is a subtle problem that usually occurs in a parallel computation when different processes/threads in the system have a different amount of work to process. This problem is especially bad for the execution that requires synchronization between processes/threads, including DL.

The computational imbalance problem that we present in this chapter is caused by high hardware resource contention. Such contention is an effect from the *lack of coordination* between different data-processing components in the DL environment, including DNN training, graph scheduling, gradient synchronization, and input pipeline components, that execute *simultaneously* and *asynchronously* on the shared hardware. Such computational imbalance degrades the DL training performance in terms of scalability and data processing throughput.

According to the scalability and computational imbalance analysis of TensorFlow with Horovod (denoted TensorFlow/Horovod), we propose four optimizations to Horovod that minimize the interactions between the data-processing components to allow them to share hardware resources more efficiently. Our first optimization, **Horovod-GS**—global sleep time optimization, solves the nonuniform sleep time problem that causes resource contention

by using a global sleep time between processes in the system. The second optimization, **Horovod-NBCS**—nonblocking cache synchronization, adopts a nonblocking communication technique to solve the resource competition problem. The next optimization, **Horovod-SCP**—static CPU resource partitioning, isolates a CPU core for handling data transfer to prevent the gradient synchronization component from interfering with other components. Our last optimization, **Horovod-TOPO**—graph topology exploitation, leverages the available computation graph topology information to delicately enhance the way that Horovod handles data transfer, thus reducing the effect of computational imbalance significantly. Our optimizations can improve the performance of various DNN trainings by up to 35% on up to 24,576 GPUs of the Summit supercomputer at Oak Ridge National Laboratory—the world's fastest super computer (as of June 2019).

6.1 Data Processing in Parallel Deep Learning

DNN learning is a complex computational method that consists of multiple data-processing components. Most modern DL frameworks assign these components to run on the available computational devices, e.g., CPUs and GPUs, simultaneously and asynchronously so as to increase resource utilization and computational throughput. Without proper coordination, however, these data-processing components compete with each other for resources, such as CPU cycles, memory bandwidth, network bandwidth, and access to the direct memory-access (DMA) engine.

In this work, we adopt a multilevel parallel DL model that provides *data parallelism* across nodes and within the node by using multiple GPUs on each node. In other words, a full replica of the DNN is trained with a different batch of input data on each GPU. Each GPU then uses *model parallelism* to further parallelize the DL training. Figure 6.1 shows the

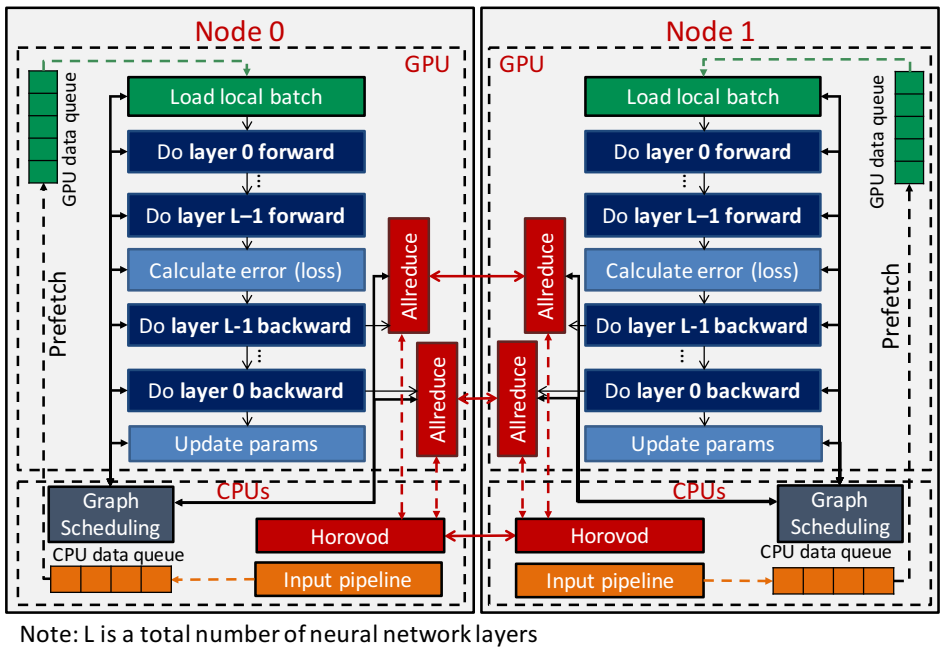

Figure 6.1: Data-processing components of DL

data-processing components of our data-parallel environment. In the figure, we show only one GPU per node for simplicity, but the actual system that we use in our experiments has multiple GPUs per node. The model comprises four main data-processing components:

1. **Graph Scheduling (occurs on the CPUs):** Each kernel/operation in the DNN is dispatched to run on the GPU by the graph scheduler that is driven by the CPU threads. Any delay in graph scheduling can slow the DNN training.

2. **Neural Network Training (occurs on the GPUs):** The core computation associated with the DNN training (i.e., forward and backward computations) occurs on the GPUs.

3. **Gradient Synchronization (occurs on both the CPUs and GPUs):** The gradient synchronization between all GPUs is performed via the Allreduce operation during the backward computation of the training. Similar to other operations, the Allre-

duce operation is dispatched to run on the GPU by the graph scheduler. (Although the Allreduce operation in Figure 6.1 is depicted as executing on the GPUs, it uses both CPU and GPU resources.) The gradient transfers are scheduled and managed by Horovod, more details of which are presented in Chapter 6.2.2.

4. **Input Pipeline Processing (occurs on the CPUs):** Input pipeline processing in the DL system involves a number of steps including file I/O, data shuffling, data augmentation, data prefetching, and host-to-device data transfer. In our experiments, we enable data batch prefetching and pipelining to avoid data-movement bottlenecks that might occur.

Of these various data-processing components, the gradient synchronization is of particular interest because of its dependence on both CPU and GPU resources. Without sophisticated coordination, it can potentially compete for both CPU and GPU resources with the other data-processing components described above. Thus, in this work, we focus on minimizing the interactions between the gradient synchronization component and the other data-processing components on the CPUs (i.e., graph scheduling and input pipeline processing).

6.2 TensorFlow/Horovod Performance Analysis

Here we profile and analyze the data processing components in TensorFlow/Horovod on a large-scale system.

6.2.1 Experimental Setup for Computational Imbalance Optimization Experiments

We first articulate our experimental setup, including our datasets, deep neural networks, supercomputing platform, data storage, DL frameworks and software stack, and experimental configuration, from which we gather our experimental data for subsequent performance analysis.

Datasets and DNNs: We use *tf_cnn_benchmarks*,[1] one of the most well-known convolutional neural network (CNN) training benchmarks. All experiments use the **ImageNet**[2] dataset—an *image classification* dataset. Our analysis and evaluations are conducted on various CNNs, including five variants of **ResNet** [63] (sizes 18, 34, 50, 101, and 152), **AlexNet** [84], **GoogLeNet** [152], **Inception-v3** [153], and **VGG16** [145].

Supercomputing platform: We use **Summit**, a supercomputer at Oak Ridge National Laboratory, as our experimental testbed. Summit has 4,608 nodes connected via Mellanox EDR 100-Gbps InfiniBand. Each node has two sockets of IBM POWER9 CPUs (total of 44 cores), six NVIDIA Tesla V100 GPUs, 512 GB of memory, and 1,500 GB NVMe (short for Non-Volatile Memory Express). Each socket connects 22 cores, three GPUs, and 256 GB of memory. For each node, two cores (one per socket) are isolated for OS tasks and cannot be used by user applications. We use six processes per node because Horovod restricts each process to drive at most one GPU. Each process has exclusive access to seven cores and one of the GPUs that is located on the same socket. Processes are limited to accessing only 256 GB of memory within their socket.

Data storage: Unlike our file I/O study presented in Chapters 3, 4, and 5, we adopt the

[1] https://github.com/tensorflow/benchmarks.git
[2] http://www.image-net.org/challenges/LSVRC/2012/

available on-node storage, NVMe, as our data storage. The dataset is staged in to NVMe prior to the execution of the benchmark. As the file I/O performance is not the main emphasis of this part of the thesis, the data staging time is not included in the performance results.

DL frameworks and software stack: We use **TensorFlow** v1.14.0-rc0 and **Horovod** v0.16.3 as our DL framework and communication subsystem, respectively. We use CUDA v10.1.168, CUDNN v7.6.1, and NCCL v2.4.7 (with GPUDirect RDMA) as the drivers for TensorFlow and Horovod. For performance, we compile the TensorFlow computation graph using the Accelerated Linear Algebra (XLA) compiler [7], which, in turn, disables any overlap between the computation and communication in the overall execution of a program. However, we do not obtain much performance improvement when the computation and communication overlap is present when XLA is disabled. Therefore, we use the XLA-enabled version as our baseline in order to distinguish the communication and computation bottlenecks from each other in our analysis.

Experimental configuration: In all of our experiments, the training is run for 500 iterations with an additional 10 "warm-up" iterations that are not included in the performance results. All our experiments use mixed-precision floating-point SGD training [112]. All experiments are run three times, and the average performance is shown.

6.2.2 Understanding Horovod and its Background Thread

The high-level overview of Horovod and its background thread can be found in Chapter 2.4.4. In this section, we explain the inner workings of Horovod that originates computational imbalance.

Tensor transfer requests associated with Horovod operations that have no dependencies with

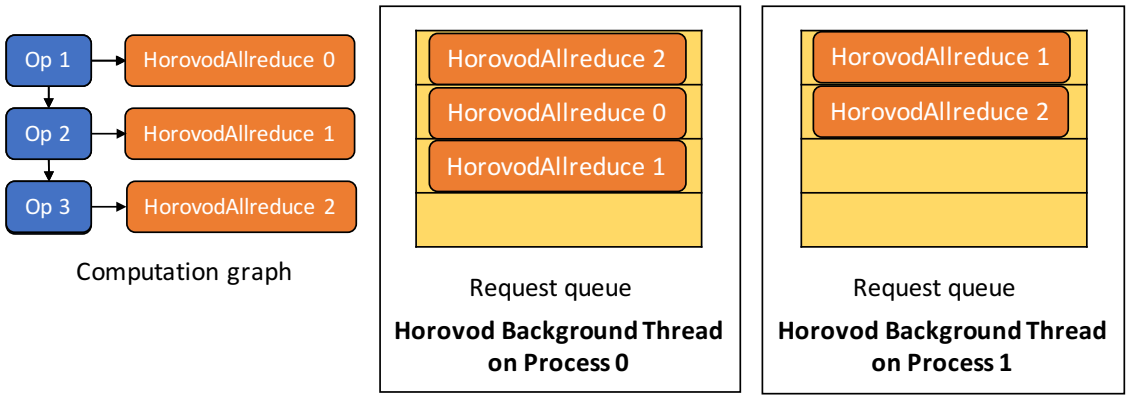

Figure 6.2: Example of a state of the Horovod request queues on two processes

one another in the computation graph could be enqueued simultaneously, for example, by different graph scheduler threads processing the graph. Therefore, even when all processes are executing the same graph, the operations in the graph, including Horovod operations, could be executed out of order, thus making the order of data transfer requests nondeterministic. Figure 6.2 demonstrates that the tensor requests in the request queues of two Horovod background threads can be different, and the requests of the same tensors can be in the different order. Because Horovod relies on other collective communication primitives, it has to ensure that data transfers for different tensors are performed in the same order on all processes. Consequently, the background threads on these processes have to perform an additional **tensor-ordering consensus protocol** to determine a globally consistent order of data transfers.

The provided implementation of the tensor-ordering consensus protocol in the Horovod background thread is unfortunately inefficient. Figure 6.3 shows the high-level workflow of this protocol. The background thread executes an infinite loop of progress checks on tensor data transfers. We refer to each such loop as a "cycle." To prevent the background thread from monopolizing a CPU core for progress checks, there is sleep time inserted between cycles. The sleep time is `HOROVOD_CYCLE_TIME` $- T_{previous}$, where `HOROVOD_CYCLE_TIME` is the max-

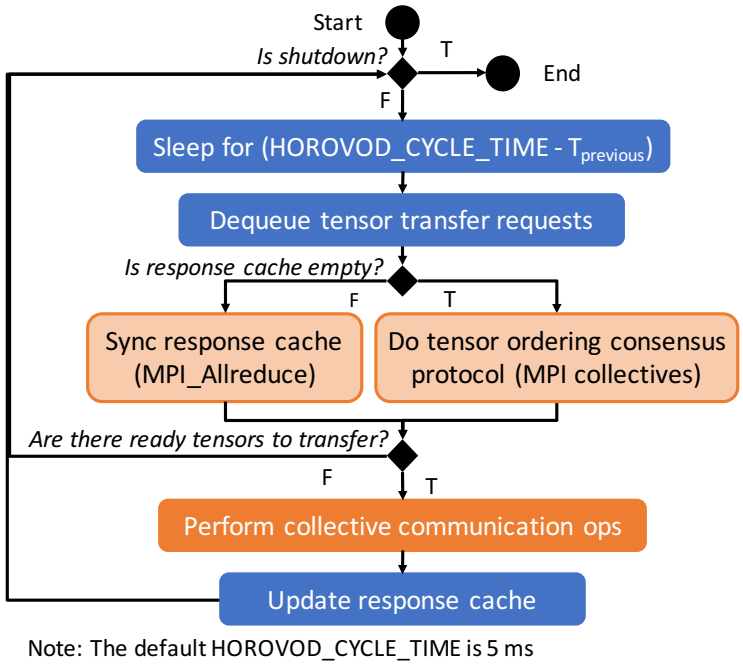

Figure 6.3: Horovod background thread workflow

imum sleep time or the maximum cycle time threshold (which is a user input; default is 5 ms) and $T_{previous}$ is the execution time of the previous cycle excluding the sleep time. If $T_{previous}$ is larger than `HOROVOD_CYCLE_TIME`, the background thread will not sleep; otherwise, it will sleep, and upon waking up, the background thread dequeues the tensor transfer requests from the request queue and attempts to create a global ordering for them through a consensus protocol.

The consensus protocol is straightforward; one of the background threads is assigned as the "master background thread" (MPI rank zero). All background threads use MPI collective operations to send the transfer request details of their ready tensors to the master background thread. The master background thread, in turn, looks through the list of ready tensors from all the background threads, forms an ordered list of tensors that are ready on all the background threads, and sends this list back to all the background threads. Once this tensor-ordering consensus protocol has completed, each background thread fuses its local tensors

and performs data transfer based on the order received from the master background thread.

This tensor-ordering consensus protocol is heavyweight and often causes severe performance degradation, especially on the master background thread. To address this issue, recent versions of Horovod (since v0.16.2) have introduced a tensor-ordering cache called a "response cache," which can be reused across cycles. This cache, which is a data structure for storing tensor information and tensor order for future use, is initially empty. Once the tensor-ordering consensus protocol occurs, each background thread locally stores the tensor request information and ordering scheme in its response cache. In the next cycle, the response cache is *not* empty, so the heavyweight consensus protocol can be avoided, but the background threads still need to synchronize their caches via `MPI_Allreduce`, to determine which tensors in the cache are ready to be transferred because this list can change from cycle to cycle as illustrated in Figure 6.4. In other words, the response cache reduces the amount of work done by the master background thread, but it does *not* (and cannot) remove the synchronization needed between the background threads.

An important aspect to understand here is that the cache synchronization is a "worst-case" requirement. Typically, the cache is *not* empty, and there are no tensors ready to be transferred. Thus, in most cycles, the background thread sleeps and then does an empty `MPI_Allreduce` for the cache synchronization. Because the arrival of tensor transfer requests is nondeterministic, each background thread still needs to participate in every `MPI_Allreduce` even if it has no new tensor transfer requests in order to prevent deadlocks.

6.2.3 Scalability Analysis

Figure 6.5 shows the results of weak scaling with TensorFlow/Horovod using the ResNet50 network (relative to linear scaling). Our baseline is XLA-enabled TensorFlow/Horovod

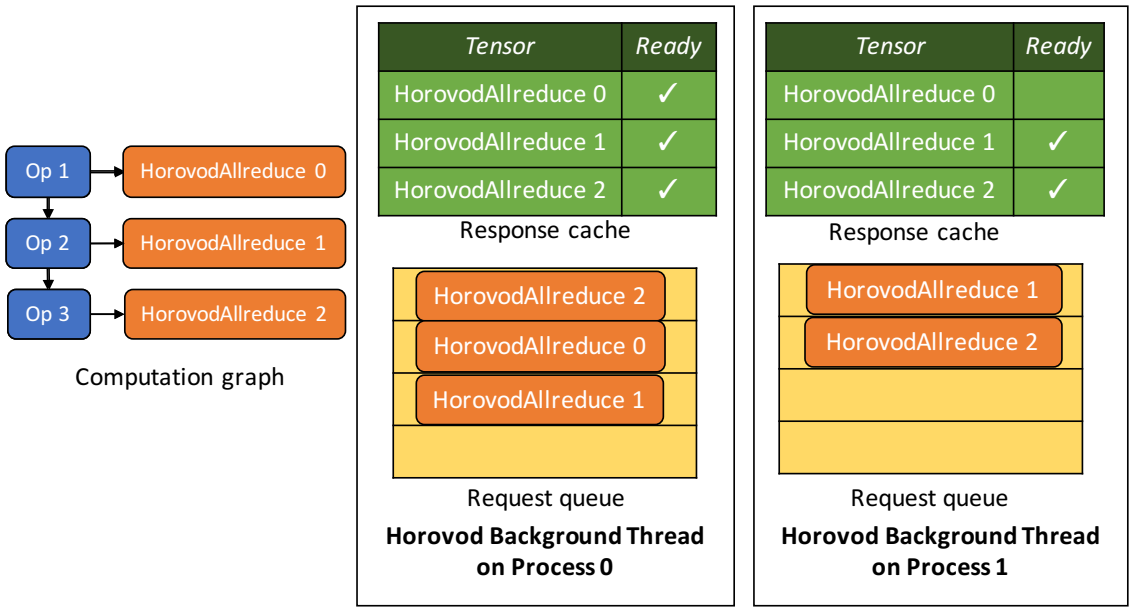

Figure 6.4: Example of a state of the Horovod request queues and response caches on two processes

(hereafter called TensorFlow/Horovod), but we show XLA-disabled TensorFlow/Horovod performance for completeness. The data-processing throughput of TensorFlow/Horovod is approximately 3.3 times *worse* than linear scaling on 24,576 GPUs; that is, the scaling loss is 69.7%.

To identify the source of this scaling loss, we profile the GPU execution, as shown in Figure 6.6, and classify the overall time into two parts: computation (denoted by "Forward & backward pass execution time") and communication (denoted by "HorovodAllreduce time"). The figure shows that the computation time stays relatively constant as the number of GPUs increase but that the communication time increases nearly linearly. In the worst case, HorovodAllreduce consumes up to 70.3% of the overall GPU execution time, which accounts for virtually all of the scaling loss noted above.

Next, we analyze the GPU time during HorovodAllreduce, as shown in Figure 6.7. The figure shows the HorovodAllreduce time separated into three parts: (1) *ncclAllReduce*, where the

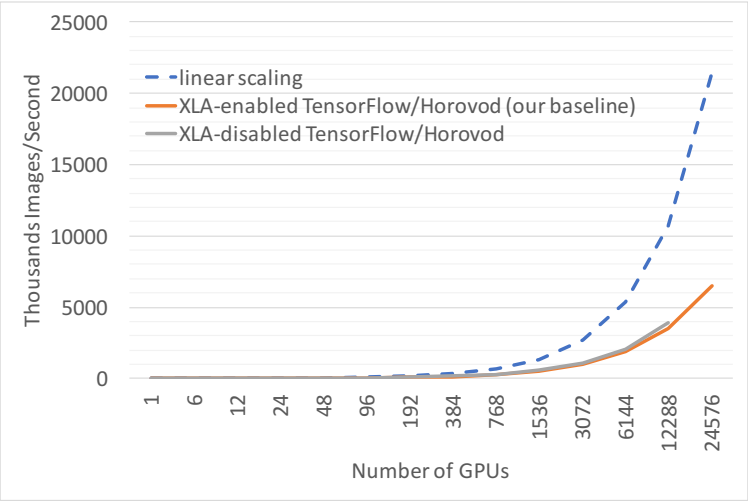

Figure 6.5: Weak scaling of TensorFlow/Horovod compared with linear scaling

GPU can be either idle (waiting for other GPUs to send data) or busy (calculating the summation of data), (2) *memory copy*, where the GPU is considered to be idle as it is using the DMA engine, but not the computation units, and (3) the *Horovod background thread overhead*, where the GPU is idle and waiting for the host to finish its work. Our profiling shows that the GPU is idle for at least 67% of the HorovodAllreduce time (i.e., summation of the background thread overhead and memory copy time). The majority of this idle time is due to the *Horovod background thread overhead*, which includes cycle latency (i.e., the sleep between cycles), tensor stalling (i.e., waiting for tensors to be ready for transfer on all processes), and tensor ordering.

6.2.4 Investigating the Horovod Background Thread

As noted in Chapter 6.2.2, the Horovod background thread spends most of its time alternating between sleeping and performing an often empty `MPI_Allreduce`. When there are tensors to be transferred, it calls the collective communication operations. The workflow of the Horovod background threads was designed for scenarios where all background threads

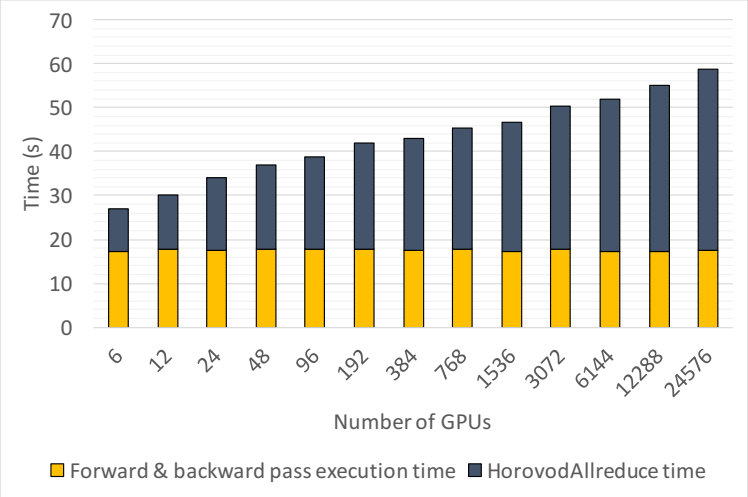

Figure 6.6: TensorFlow/Horovod GPU time breakdown. (We note that using XLA disables any overlap between the computation and communication, as explained in Chapter 6.2.1)

are fairly synchronized. In such a scenario, as shown in Figure 6.8, the background threads spend most of their time sleeping. Thus, they would not compete for resources with other data-processing components, and consequently, would not create any further performance imbalance in the computation. In reality, however, this is not always the case.

In real DNN training, even when all processes are computing on exactly the same computation graph, there can be a slight imbalance in their execution time or the state of the various threads in the system (e.g., which threads are executing at a given point in time). Such imbalance is expected but generally small and uninteresting. However, the *cascading effect* of such small imbalances is of particular interest as it makes up virtually all of the HorovodAllreduce time.

Consider a case with two processes, where both processes are computing on the same computation graph, but the state of the execution or that of the various threads is not exactly identical on both processes. For these processes, when the background threads are ready to be scheduled by the OS, the two background threads might have to wait for vastly different amounts of time to get scheduled. This difference in actual scheduling time depends

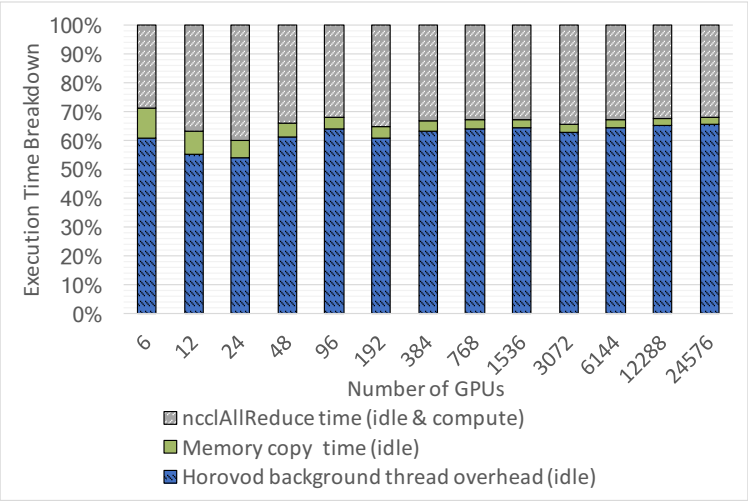

Figure 6.7: TensorFlow/Horovod HorovodAllreduce GPU time breakdown

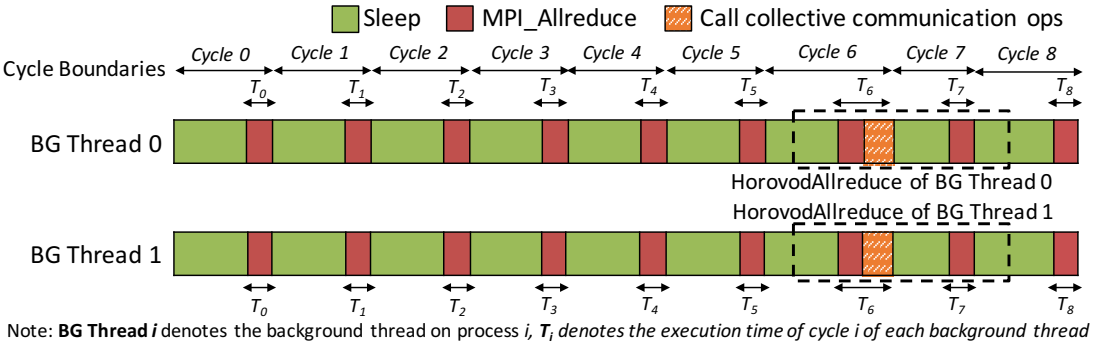

Figure 6.8: A perfect synchronization of Horovod background threads

on when the OS decides to preempt the other currently executing threads (i.e., the input pipeline and graph scheduling threads) and to execute the background thread. This wait time for preemption can be as high as tens of milliseconds on modern Linux versions. We call this scenario, where some background threads take longer to be scheduled than the other background threads, as "oversleep."

Figure 6.9 demonstrates the cascading effect of computational imbalance that propagates from one cycle to the next. In the figure, `BG Thread i` denotes the background thread on process `i`. In *Cycle 0*, `BG Thread 0` arrives at the `MPI_Allreduce` function first and

consequently takes longer to complete the operation because it is waiting for `BG Thread 1`, that oversleeps, to call `MPI_Allreduce`. In this case, `BG Thread 1` is a **straggler** thread.

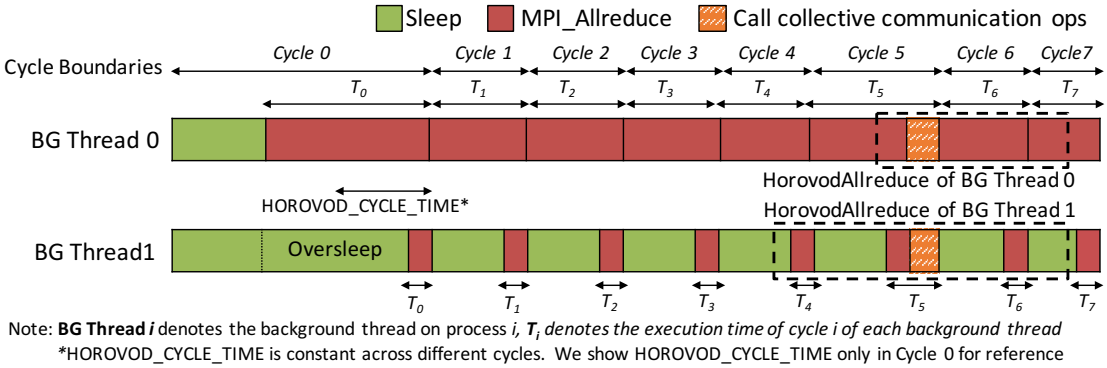

Figure 6.9: Horovod background thread oversleep problem

In the next cycle (i.e., *Cycle 1*), `BG Thread 0`'s previous cycle time T_0 is larger than the maximum cycle time threshold, `HOROVOD_CYCLE_TIME`, and thus Horovod would not let it sleep at all, as described in Chapter 6.2.2. `BG Thread 0` would then issue `MPI_Allreduce` right away. In contrast, T_0 of the straggler thread is smaller than `HOROVOD_CYCLE_TIME` causing it to sleep in *Cycle 1*. This action would cause the straggler thread to be delayed in reaching its `MPI_Allreduce` in the next cycle as well, further exacerbating the computational imbalance impact.

In addition, while `MPI_Allreduce` waits for other processes to arrive, it **spin waits**, thus consuming CPU cycles and potentially slowing down other data-processing components, namely, input pipeline processing and graph scheduling. For input pipeline processing, the impact is minimal because the outcome of the input pipeline is used in the next training iteration (recall that data prefetching is enabled), and the delay does not stall the current iteration. For graph scheduling, however, this slowdown can cause the forward and backward computation on the GPU to be delayed. This delay causes some GPUs (e.g., the GPU associated with process 0 in Figure 6.9) to execute the gradient synchronization late, thus

making the imbalance show up in the HorovodAllreduce time as the Horovod background thread overhead.

6.2.5 Resource Contention Analysis

As explained in Chapter 6.2.4, when computational imbalance are present, the Horovod background thread spends the majority of its execution time busy-waiting inside `MPI_Allreduce`. Thus, we expect the user-space time of the background thread to be the dominant portion of the execution time. We measure the CPU usage of the background thread during the computation as shown in Figure 6.10. As expected, the background thread sleeps for a very small amount of time and spends most of its time in user space. These results indicate that the background thread is active and occupies the CPU core for 60–90% of the execution time, which is unusually high considering that its workflow is not computationally expensive.

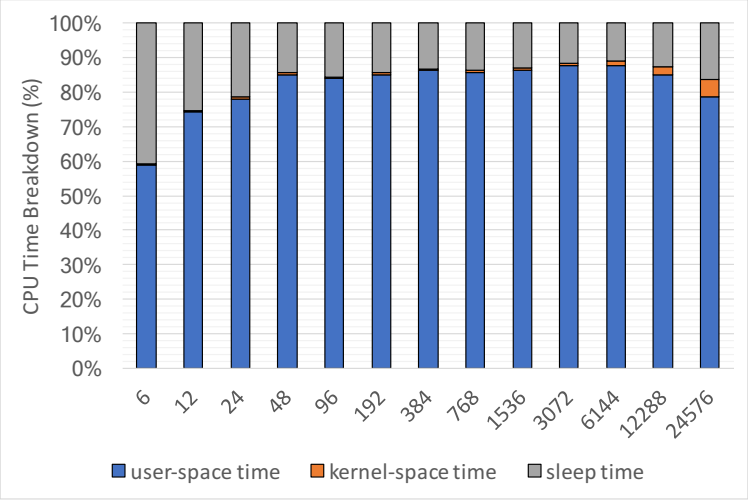

Figure 6.10: Horovod background thread's CPU usage

We also analyze the involuntary context switches (ICSWs) in the DL data-processing components to determine whether the background thread is actually competing for CPU time. ICSWs occur when a thread is suspended by the OS scheduler in favor of other threads either

to maintain fairness in the CPU scheduling policy or when other higher-priority threads exist. Typically, ICSWs happen when the system lacks CPU resources. A large number of ICSWs can indicate high resource contention. We compare the ICSWs of the benchmark with two data-processing configurations: (1) using synthetic data and (2) using real ImageNet data. We note that synthetic data is generated on the GPU; thus the input pipeline processing does not exist in the synthetic data execution. This would result in less CPU contention for the synthetic data if there indeed is contention. (Performance of the synthetic data is up to ∼30% better than the real ImageNet data.) We notice that the number of ICSWs of the real data execution is ∼8.5-times higher than that of the synthetic data execution. This indicates that high resource contention does occur when the input pipeline and the Horovod background thread are present at the same time.

6.3 Design and Implementation of Computational Imbalance Optimizations

In this section, we present four solutions to alleviate computational imbalance problem in distributed DL processing, based on our performance analysis of TensorFlow/Horovod in Chapter 6.2.

6.3.1 Horovod-GS: Global Sleep Time Optimization

The oversleeping of some background threads causes the processes to issue `MPI_Allreduce` at different points in time, as noted in Chapter 6.2.4; in other words, the processes are "out of sync" in calling `MPI_Allreduce`. To address this issue, we propose *Horovod-GS*. In *Horovod-GS*, each background thread, instead of using just its local knowledge to figure out how long

it needs to sleep, uses a **globally coordinated sleep time** to ensure that all processes use the same sleep time every N cycles. This prevents the computational imbalance effect from propagating beyond N cycles. The intent here is to separate the imbalance from the actual data transfer time in `MPI_Allreduce`.

Before sleeping, each background thread computes its local sleep time using the same formula as in the original Horovod (`HOROVOD_CYCLE_TIME`$-T_{previous}$). Then, all processes determine the globally minimum sleep time by using another `MPI_Allreduce`. Once the minimum sleep time is received, each background thread sleeps for the globally minimum sleep time. The rest of the workflow is the same as the original Horovod. This approach prevents the computational imbalance effect from propagating to subsequent cycles.

Figure 6.11 shows an example of the timeline of the background threads in Horovod-GS. Suppose we resynchronize the background threads every ten cycles. From the figure, `BG Thread 1` oversleeps in *Cycle 0*. As explained in Chapter 6.2.4, the local sleep times in *Cycle 1* of `BG Thread 0` and `BG Thread 1` are different (i.e., `BG Thread 0`'s local sleep time is zero while `BG Thread 1`'s local sleep time is nonzero). Since we have the background threads synchronize their sleep times before actually sleeping, both background threads will not sleep in *Cycle 1* in this example. Here, the computational imbalance effect is not transferred to the subsequent cycles.

As noted above, the second `MPI_Allreduce` that we introduce in *Horovod-GS* occurs once every N cycles. So, theoretically, a smaller value for N creates higher synchronization overhead while reducing the imbalance effect, whereas a large value of N provides the opposite tradeoff. Empirically, we found that $N = 1$ delivered the best performance on our system—although, depending on the dataset and the system, this value may need to be tuned appropriately.

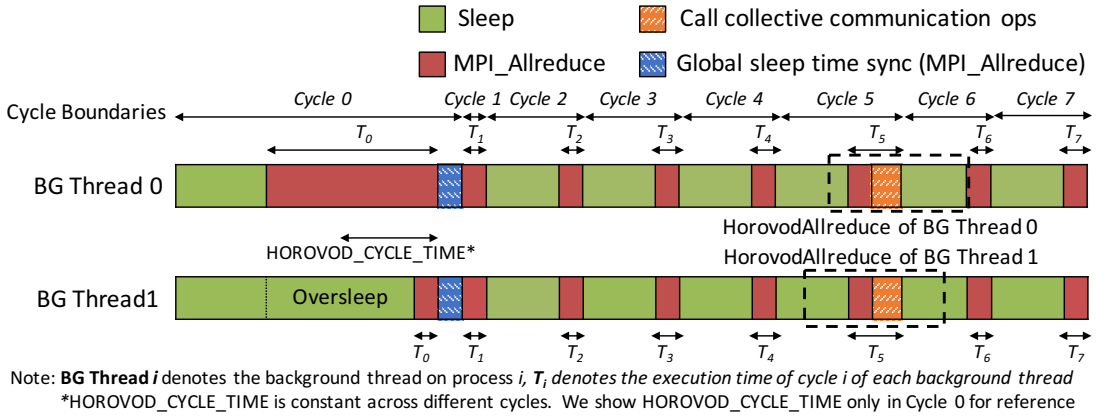

Figure 6.11: Timeline of Horovod-GS's background threads

6.3.2 Horovod-NBCS: Nonblocking Cache Synchronization

While Horovod-GS can reduce the computational imbalance effect, it cannot completely prevent the background thread from competing for resources with other components. Specifically, the imbalance in the first `MPI_Allreduce` still remains and typically consumes the most time. Thus, the Horovod background thread still spends a significant amount of time occupying the CPU cores and competing for resources with the input pipeline and the graph scheduling components.

Thus, we propose Horovod-NBCS (i.e., nonblocking cache synchronization), where we seek to limit the time spent inside `MPI_Allreduce` in order to free up computational resources for the other two data-processing components. To do so, we leverage nonblocking MPI collective operations, specifically, `MPI_Iallreduce` and `MPI_Test`, for the response cache synchronization. While this approach does *not* avoid the out-of-sync problem between processes, the processes no longer compete for resources with other components because the time spent inside each MPI call is finite (as guaranteed by the MPI standard for all nonblocking operations) and typically small.

Figure 6.12 shows an example of the timeline of the background threads in Horovod-NBCS.

From the figure, the background threads are still out of sync (i.e., `MPI_Iallreduces` are issued at the different points in time between the two background threads), however, the nonblocking communication prevents the background threads from occupying the cores (i.e., "sleep" is the dominant portion in the timeline). Note that in the implementation, we invoke `MPI_Test` in a loop until the message arrives. There is a small constant sleep at the end of the loop (denoted "Sleep between MPI_Tests" in Figure 6.12) to prevent the background thread from monopolizing the core.

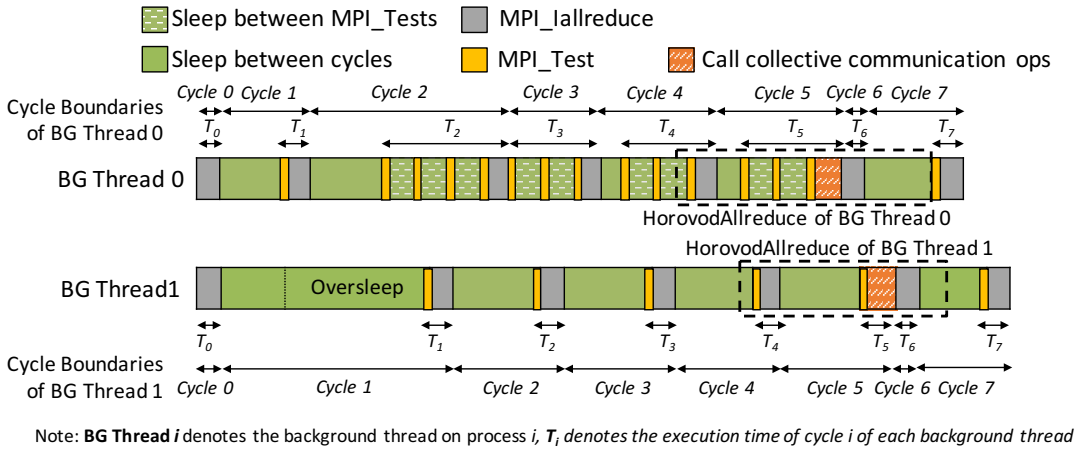

Figure 6.12: Timeline of Horovod-NBCS's background threads

The Horovod-GS and Horovod-NBCS solutions are applicable only for cases where the time taken by the Allreduce operation is small enough that it is completely overlapped by the computation time. If this balance changes, then the computation would no longer be able to fully hide the communication and imbalance time and would result in performance degradation.

6.3.3 Horovod-SCP: Static CPU Resource Partitioning

With Horovod-SCP, we address the resource contention problem via a simple static partitioning of resources. Specifically, to avoid contention between the different data-processing components, we partition the available cores into groups such that each group of threads

that executes a different data-processing component gets a different set of cores. This guarantees that there is no contention between the different data-processing components, thus potentially alleviating computation time imbalance.

As we will see in Chapter 6.4, Horovod-SCP successfully reduces the contention between different data-processing components to alleviate computational imbalance. Despite the impressive performance gains, however, we view Horovod-SCP as a somewhat of a workaround. Specifically, while Horovod-SCP does alleviate the biggest cause for load imbalance, it comes with several shortcomings.

First, the static partitioning of CPU resources means that any variation in processing needs that arise during the execution of the DL workflow cannot be dynamically resolved. For instance, the background thread only needs to be active for a small part of the total execution, but having a dedicated core means that that core cannot be used for other data-processing components when the background thread is idle. This can impact the overall performance if the other data-processing components starve for CPU resources. Second, even with a dedicated core, computation time imbalance cannot be fully avoided. This is because, as described in Chapter 6.2.4, even when all processes are computing on exactly the same computation graph, there can be a slight imbalance in their execution time. Because of this slight imbalance, how long each background thread spends in the `MPI_Allreduce` can be different, which would cause different threads to sleep for different amounts of time in the next cycle, which would further increase the imbalance. Thus, future work will study and integrate dynamic partitioning, as appropriate.

6.3.4 Horovod-TOPO: Graph Topology Exploitation

While the previous solutions can help reduce the time the background thread spends competing for computational resources, they are still fundamentally limited by the way Horovod performs tensor ordering. In particular, they rely on the most generic possibility where the tensor transfers can be issued in any arbitrary order. However, this is not true in reality and over-generalizes the TensorFlow workflow.

TensorFlow uses a graph processing workflow, and the order in which tensors are issued depends on the graph structure. Tensors that are *logically concurrent* (i.e., belong to graph nodes with no dependency between them—for example, *Allreduce*0, *Allreduce*1, and *Allreduce*4 in Figure 6.13) can be issued in any order. When Horovod sees such a tensor request, it can wait for the other logically concurrent tensor requests to be issued without creating deadlock. In contrast, for two tensors whose corresponding graph nodes have a dependency between them (for example, *Allreduce*0, *Allreduce*2, and *Allreduce*3 in Figure 6.13), there is a guaranteed ordering where the second tensor cannot be issued before the first tensor operation has completed.

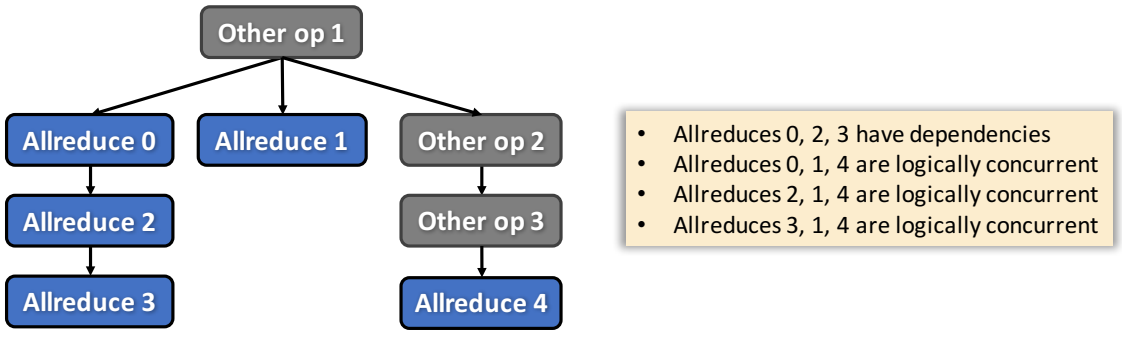

Figure 6.13: Example of TensorFlow computation graph

Thus, our Horovod-TOPO solution seeks to eliminate the original tensor ordering and the response cache synchronization by performing a one-time TensorFlow graph analysis. The core idea of Horovod-TOPO is to analyze the TensorFlow graph and the dependencies between

the graph nodes to form a **partial logical ordering** of tensor data-movement requests. The generated ordering is a *logical ordering* because some tensor requests are logically concurrent and can be issued in any arbitrary order by the graph scheduling threads. The generated ordering is *partial* because the graph dependencies restrict the reordering of some tensor requests have dependencies between one another. Thus, their tensor transfers cannot be reordered). Based on this partial logical ordering, we can then precompute a **tensor fusion scheme** that determines which tensors can be fused together so that the data-transfer requests can be larger, thus amortizing data-transfer overhead. Once the tensor fusion scheme is determined, it is stored within Horovod and utilized for all future computation iterations. Thus, this topological graph analysis needs to be done only once and never repeated.

In TensorFlow, normally only a subgraph is executed at any given time. A subgraph is identified by the user with "fetches," which are nodes in the graph whose outputs will be obtained from the execution, i.e., fetches are sink nodes of the subgraph. On the first execution of the subgraph, we assign an identification number (ID) to every Horovod operation in the subgraph. We then adopt a traditional reverse depth-first search algorithm to traverse from fetches to the root nodes to identify all Horovod operations and their dependencies. The time complexity of this algorithm is $O(V+E)$, where V and E are the number of nodes and the number of edges in the subgraph, respectively. The ID represents the chronological order in which the operations are executed. To account for operation dependencies, parent operations are assigned a smaller ID than are children operations. Among sibling operations, IDs are assigned based on the order that they are added to the graph. For parallel nonsibling operations, we assign IDs to the operations according to their depth in the subgraph. (If the depths are the same, the ID assignment is arbitrary.)

Together with the tensor order, we determine the tensor fusion model (i.e., which tensors should be fused before communicating) the first time that a subgraph is executed. We follow

Horovod's original approach to fuse only HorovodAllreduce's tensors and to cap the fusion buffers at `HOROVOD_FUSION_THRESHOLD` (64 MB by default).

Figure 6.14 shows an example of our tensor ordering and tensor fusion. The fetches in this example are *Allreduce*1, *Allreduce*3 and *Allreduce*4. From the figure, we assign the parent operations to have a smaller ID than their children (e.g., *Allreduce*0 and *Allreduce*2). The parallel nonsibling operations are assigned IDs based on their depth in the subgraph (e.g., *Allreduce*4 has a larger ID than *Allreduce*1). *Allreduce*0 and *Allreduce*1 can be fused as they are logically concurrent, likewise for *Allreduce*3 and *Allreduce*4. In contrast, *Allreduce*0 and *Allreduce*2 cannot be fused because they share a dependency. Likewise, *Allreduce*2 and *Allreduce*3 have to be in different fusion buffers.

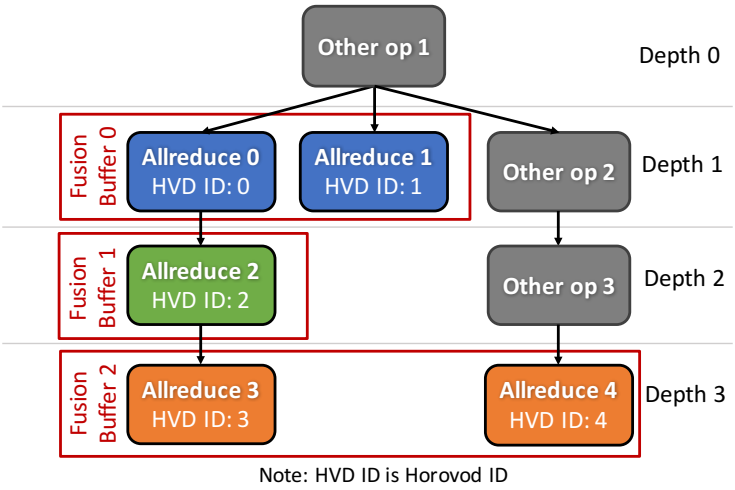

Figure 6.14: Example of tensor ordering and tensor fusion

During the actual execution of the TensorFlow graph, the background thread performs a tensor transfer only if the transfers of all tensors with smaller IDs have been issued as shown in Figure 6.15. We use one request array per fusion buffer for storing tensor transfer requests from the TensorFlow's graph scheduler threads. Once a Horovod operation is executed, a graph scheduler thread puts a tensor transfer request into the designated slot (i.e., based on the Horovod ID) in the request array. The background thread repeatedly checks whether

the request at the current slot has been added into the array. Once the request arrives, the background thread determines whether this request is for the last tensor in the fusion buffer. If it is not, the background moves onto the next slot. If it is, the background thread fuses all tensors in the request array, issues the corresponding data transfer for this request array (by calling the appropriate collective communication function), and shifts to the next request array. It then repeats the same steps until all requests have been issued. After all tensor transfers have been issued, the Horovod background thread goes to sleep and does not wake up until the next tensor transfer request arrives. The queues and variables that are shared between the different threads are managed by using C++11 std::atomics, with some portions of the code optimized for IBM POWER9 CPU hardware atomics and memory ordering/consistency semantics.

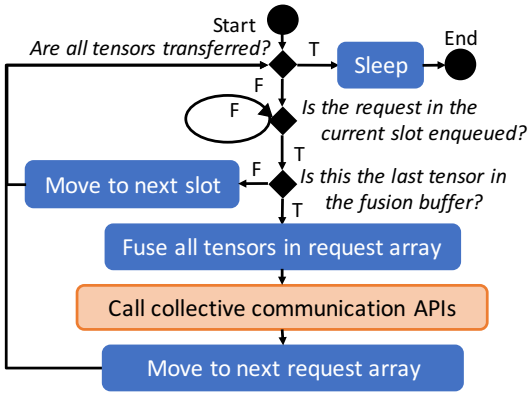

Figure 6.15: Horovod background thread's workflow in Horovod-TOPO

While Horovod-TOPO cannot guarantee that the data transfer requests for logically concurrent tensors will always *arrive* in the same order, it does guarantee that (1) the data transfer requests are *issued* in the same order on the different background threads and (2) the background threads do not have to wait indefinitely before issuing a data transfer request. Background threads wait for additional tensor requests to be issued only when the corresponding graph nodes are logically concurrent, and thus, the difference in their arrival is bounded by a finite amount of time.

6.4 Computational Imbalance Optimization Experiments and Results

We evaluate the performance of our proposed solutions and compare them against that of the original TensorFlow/Horovod implementation.

6.4.1 Evaluation of Proposed Solutions on ResNet50 Training

We first measure the weak-scaling performance of our various solutions using the ResNet50 network and the ImageNet dataset. In this experiment, we use a fixed local batch size (i.e., number of samples per GPU in one iteration) of 32; the global batch size increases proportionally with the number of GPUs. Figure 6.16(a) shows the data-processing throughput for the different approaches (i.e., original Horovod and our four optimized versions, namely GS, NBCS, SCP, and TOPO) and Figure 6.16(b) shows the improvement percentage compared with TensorFlow/Horovod. All four optimizations outperform TensorFlow/Horovod by up to 10%, 16%, 18%, and 21%, respectively. Despite using each of the four techniques individually, we combine SCP with TOPO (denoted SCP-TOPO) to verify the combinatory benefit of the two techniques. We observe that SCP-TOPO outperforms TOPO in some cases yielding up to 23% performance improvement over the original Horovod. Technically, the performance of TOPO should be comparable or better than the performance of SCP-TOPO since all CPU cores in TOPO are shared among all data-processing components. However, it is highly possible that the Horovod background thread in TOPO is interfered by the other threads in the system causing it to not be able to handle the tensor transfers as fast and as efficient as the background thread in SCP-TOPO that has its own core.

Figure 6.16(a) also shows that for runs with the number of GPUs $\geq$ 3,072, our global batch

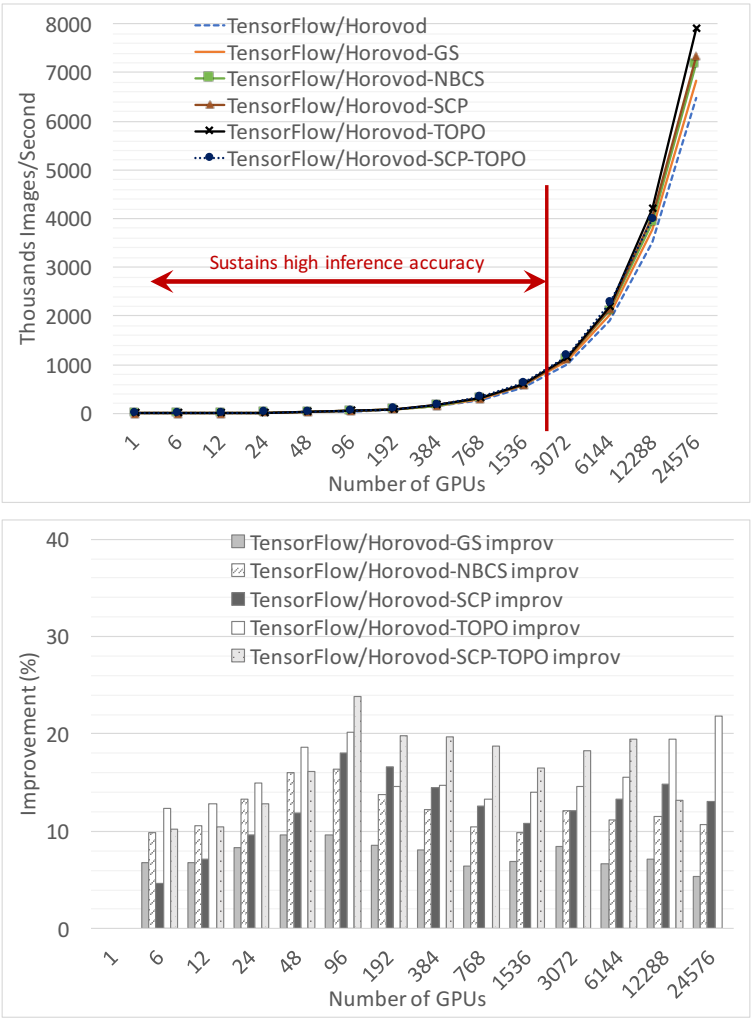

Figure 6.16: Weak-scaling results on ResNet50 on Summit: (a) image-processing rates (images/second); (b) percentage improvement in performance

size becomes large enough that the inference accuracy drops. (We can sustain the state-of-art inference accuracy of 75% until up to the global batch size of 61,440.) Despite this drop in accuracy, we highlight the following two points: (1) our proposed solutions are also applicable to smaller global batch sizes and deliver significant performance improvements even in such cases, and (2) the general trend in the research community seems to be towards algorithmic improvements that allow for larger batch sizes, thus indicating the increasing importance of studying the scalability of DL frameworks on large supercomputing systems.

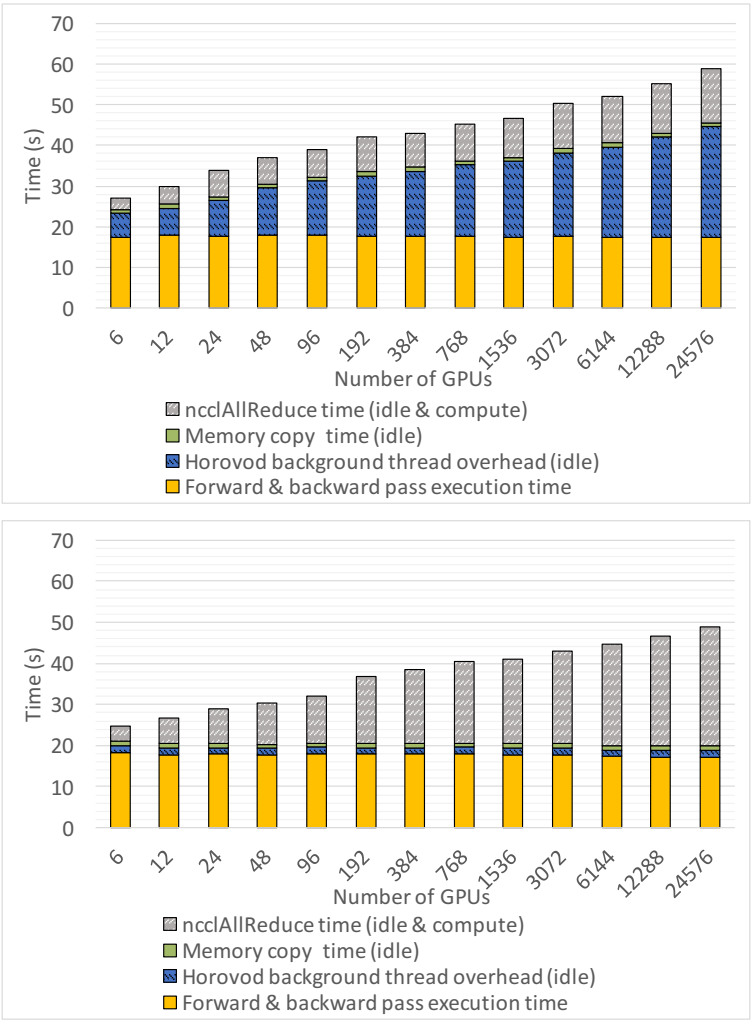

Figure 6.17: GPU time breakdown of ResNet50 training: (a) TensorFlow/Horovod; (b) TensorFlow/Horovod-TOPO

Because TensorFlow/Horovod-TOPO delivers the best performance gain, we further analyze its performance in order to understand the improvement. Specifically, we compare the GPU time breakdown of TensorFlow/Horovod-TOPO with that of the original TensorFlow/Horovod in Figure 6.17. The improvement with TensorFlow/Horovod-TOPO is mainly from the reduction of the Horovod background thread overhead. On 24,576 GPUs, this overhead shrinks from ∼46% of the execution time to 3.4%. We note that some computation time imbalance still remains in the execution, as evidenced by the increase in the time taken by

ncclAllReduce. This imbalance does not appear as part of the Horovod background thread overhead because the background thread's workflow is now completely nonblocking. Instead, it shows up in the ncclAllReduce time. Investigating this imbalance is outside the scope of this work but part of our future work.

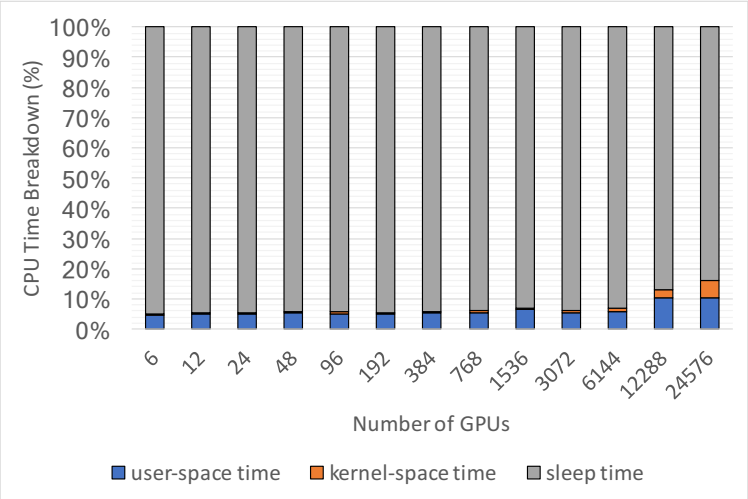

Figure 6.18: Horovod-TOPO background thread's CPU usage

We also measure the CPU usage time and involuntary context switches in TensorFlow/Horovod-TOPO. The background thread in TensorFlow/Horovod-TOPO is active only for ∼6% of the execution time (as shown in Figure 6.18), which is much smaller than the 60–90% of the execution time in TensorFlow/Horovod. Similarly, the involuntary context switches in TensorFlow/Horovod-TOPO are 3–25 times smaller than those in TensorFlow/Horovod. These results demonstrate the ability of TensorFlow/Horovod-TOPO to almost entirely eliminate the resource contention that is caused by the Horovod background thread.

Figure 6.19 presents our strong-scaling results. Here we use a fixed global batch size of 24,576, which still sustains state-of-the-art accuracy. The local batch size is scaled proportionally with the number of GPUs. We use at least 96 GPUs in this experiment to ensure sufficient memory. The performance results show similar trends as weak scaling: our optimizations improve performance by up to ∼19%. We experience a slight drop in the image processing

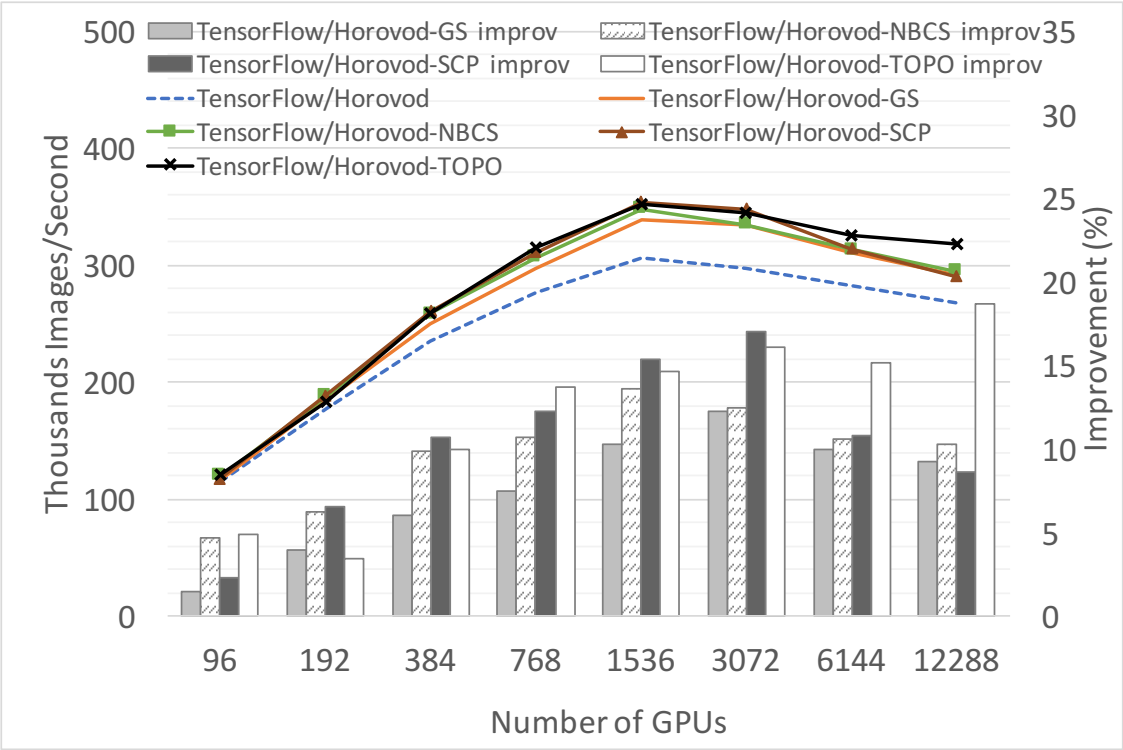

Figure 6.19: Strong-scaling results on ResNet50 on Summit: image-processing rates (images/second) and percentage improvement in performance

rate after 1,536 GPUs because the local batch size becomes too small (i.e., ≤ 8) and the communication time, which increases with the number of GPUs, dominates the overall time.

6.4.2 Horovod-TOPO's Performance on Other Neural Networks

Below we analyze the performance of Horovod-TOPO while training a variety of neural networks.

Graph Parsing Overhead

As noted in Chapter 6.3.4, Horovod-TOPO traverses the TensorFlow computation subgraph prior to the first execution of the subgraph to obtain tensor dependencies and the tensor

fusion model. To understand how much impact the graph parsing has on the overall performance of training various neural networks, we compute the node and edge counts in the subgraph and measure the graph traversal overhead, as shown in Table 6.1.

Table 6.1: Computation graph characteristics and Horovod-TOPO's graph traversal overhead

CNN Name	Node Count	Edge Count	Traversal Overhead	
			ms	Percent[a]
ResNet18	1,714	2,565	32	0.18
ResNet34	2,866	4,325	86	0.37
ResNet50	3,988	6,042	159	0.04
ResNet101	7,558	11,499	594	1.49
ResNet152	11,128	16,956	1,274	2.33
AlexNet	655	893	6	0.02
GoogLeNet	3,499	4,970	182	0.89
Inception-v3	6,146	9,211	382	1.12
VGG16	1,091	1,479	17	0.03

[a]Percentage in the execution time of the complete ImageNet training on 12,288 GPUs.

As expected, the graph traversal time increases proportionally with the node and edge counts. In most cases, the total traversal overhead is a few tens or hundreds of milliseconds. The overhead is the highest for ResNet152, which takes around 1.2 seconds for the graph traversal. To put this in perspective, the overall execution time of the training runs is typically on the order of tens of minutes to even hours. For ResNet152, for example, our graph traversal overhead accounts for a mere 2.33% of the total execution time of a complete ImageNet training. Thus, we conclude that Horovod-TOPO incurs an insignificant amount of overhead for the graph parsing.

Weak-scaling Evaluation

Figure 6.20 illustrates the image-processing rates and improvement percentage while using TensorFlow/Horovod-TOPO compared with TensorFlow/Horovod for various neural net-

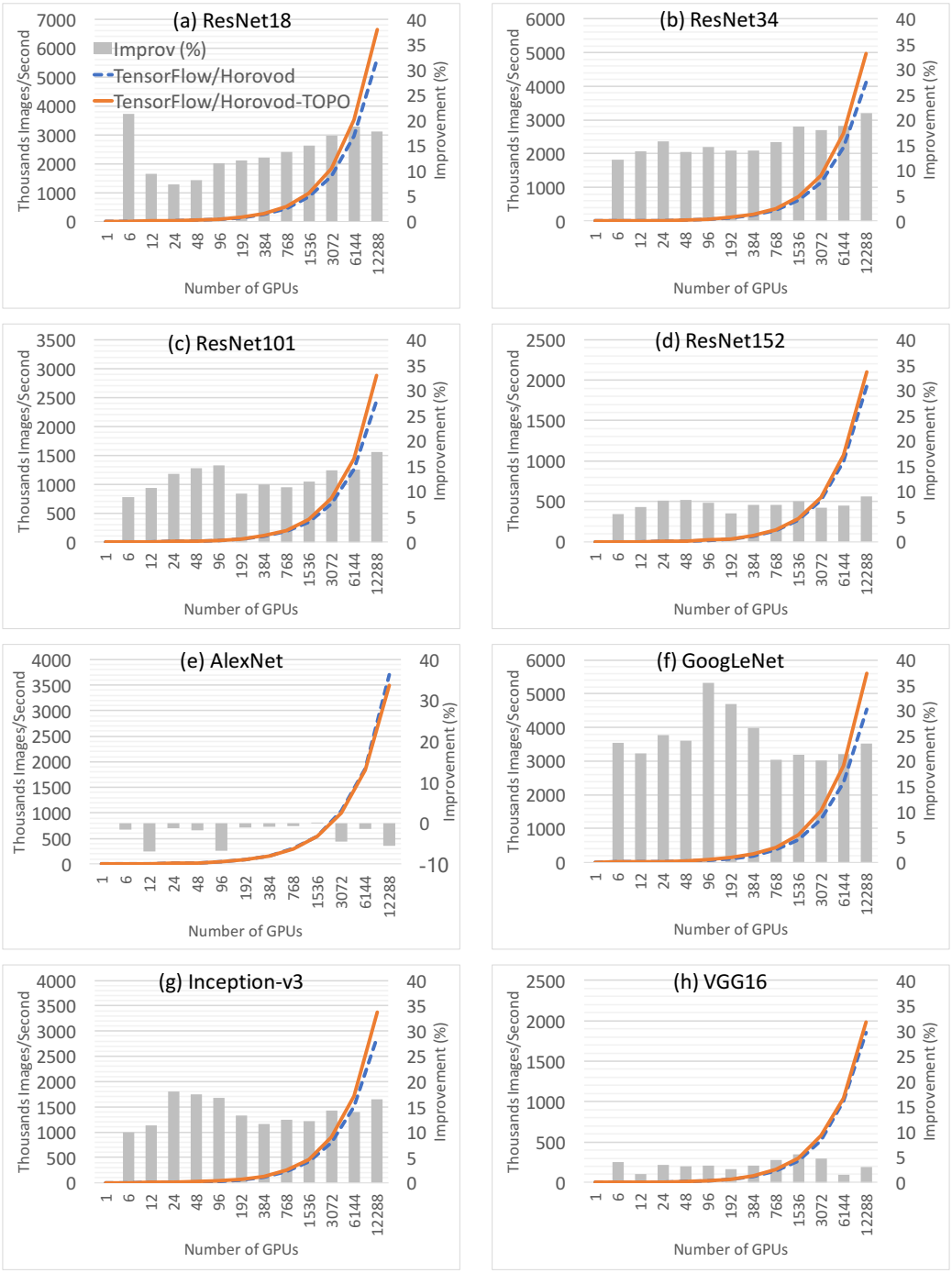

Figure 6.20: Weak-scaling results on various DNNs (image-processing rates and improvement percentage): (a) ResNet18; (b) ResNet34; (c) ResNet101; (d) ResNet152; (e) AlexNet; (f) GoogLeNet; (g) Inception-v3; (h) VGG16. Note: the scale of the image-processing rate axis varies among graphs

works. For all the experiments, the input pipeline is identical. For simplicity in discussion, we define the term "**computational imbalance susceptible window**" or "**CIS window**," which refers to the period where the graph scheduling component and the input pipeline are both active but tensor transfers have not yet occurred. During the CIS window, the computational imbalance have the highest impact on the graph scheduling, which in turn has a direct impact on the core neural network training on the GPUs. When the input pipeline is not active, the computational imbalance do not have a significant impact on the graph scheduling because the cores are mostly free. In our evaluation, for the ResNet networks, we observe 5–21% performance gain. The smaller ResNets tend to achieve larger improvement because the input pipeline computation amount is fixed and thus smaller ResNet networks have a larger CIS window (as a fraction of the total execution time) than the larger networks. The CIS windows of ResNets 18, 34, 50, 101, and 152 are approximately 46%, 55%, 48%, 30%, and 23%, respectively.

Similar to ResNets, DNNs that have a large CIS window tend to show better performance improvements. For example, GoogLeNet, which has an CIS window of 56%, achieves the best performance improvement with an average improvement of 25% and a maximum improvement of 35%. Inception-v3 has a slightly smaller CIS window of 42% and correspondingly achieves a smaller performance gain of 10–18%. VGG16 has a relatively small CIS window of 32%, limiting its performance gain to 2–6%.

For AlexNet, Horovod-TOPO achieves slightly worse performance (∼6%) than the original Horovod. This is because the computation graph of AlexNet is *very small* (see Table 6.1) and the overhead associated with managing the tensor ordering and fusion buffers hurts performance more than the benefit of the reduced cache synchronization. While additional (engineering) optimizations to Horovod-TOPO to improve how the tensor ordering and fusion buffers are managed could be made, they would simply bring the performance of Horovod-

TOPO in line with that of original Horovod.

6.5 Chapter Summary

We investigated the limitations in scaling DL frameworks to large-scale supercomputing systems. Specifically, we analyzed TensorFlow and Horovod—state-of-the-art DL software frameworks—and identified resource contention between data-processing components, which causes straggler processes or computational imbalance, as the root cause of their scaling limitations. To address this scaling limitation, we proposed four solutions that efficiently tackle such computational imbalance problem and demonstrate up to 35% improvement in performance on 24,576 GPUs of the Summit supercomputer at Oak Ridge National Laboratory.

Chapter 7

Related Work

This chapter discusses work related to our work, including DL frameworks, file I/O optimizations, communication optimizations, and algorithmic improvement to parallel DL.

7.1 Deep Learning Frameworks

Caffe is a well-known DL framework for which a number of parallel derivatives have been proposed. Most of its derivatives [3, 15, 92], focus on parallel efficiency improvements of the training, but only in the computation and communication aspects. In these frameworks, file I/O and computational imbalance aspects are left untouched. Apart from Caffe, other open-source DL frameworks have been developed, including Google TensorFlow [6, 166], Theano [105, 161], Facebook Caffe2 [148], PyTorch [79], Microsoft Cognitive Toolkit (CNTK) [135], Apache MXNet [21], and Chainer [164]. These frameworks provide different competitive advantages in terms of training features and platform compatibility. All of them support at least one parallel model shown in Chapter 2.3. Moreover, most of them can dispatch threads or even whole CPUs (if the system is equipped with accelerators) for communication and I/O prefetching and preprocessing. Such prefetching techniques, however, can hide some of the I/O and communication costs when the costs are smaller than that of computation, but they cannot fully avoid it. We will discuss details of these frameworks in subsequent sections.

7.2 File I/O Optimizations

In this section, we present various file I/O optimization techniques for parallel DL ranging from file I/O frameworks, storage architecture, and I/O pipelining. We compare and contrast our file I/O system to others to provide the insight on how our file I/O optimizations can be applied and generalized for most existing file I/O system.

7.2.1 File I/O Subsystems in Deep Learning Frameworks

Most DL frameworks adopt a core I/O infrastructure similar to that of Caffe in order to perform parallel data I/O. For instance, Caffe2 inherits the I/O subsystems from Caffe. Thus, its distributed I/O subsystems are highly similar to the parallel extensions of the Caffe framework that we used in our file I/O optimization work. PyTorch supports a broad range of data formats, the most popular of which is NumPy [121]. Both the memory and file layouts of NumPy can be irregular. For example, bytes of a single array can be laid out into noncontiguous chunks of a file or memory. Since the file structure is not deterministic, NumPy supports partial database access via `mmap`, the same as LMDB, in order to avoid reading the entire file to memory. To the best of our knowledge, there is no other way to partially load NumPy data from a file without using `mmap`, thus making it susceptible to the same shortcomings as LMDB. TensorFlow's I/O subsystem, by default, performs replicated data reads across different processes, but such a model can hurt the accuracy of the training because of reduced diversity of the sample data across different processes. Data sharding, which would make its data processing equivalent to that of Caffe, can be enabled through its high-level API to filter out unwanted data. While data sharding improves TensorFlow's accuracy, however, it also causes extra and redundant data access between processes similarly to what LMDB suffers from.

In summary, while our file I/O optimization work uses Caffe for the experiments, we believe that the lessons learned are generally applicable to other frameworks, too. In fact, a common practice in the community is to store datasets in LMDB format as it is natively supported by various other well-known DL frameworks such as TensorFlow, Caffe2, PyTorch, and Keras-TensorFlow. While other database formats certainly exist, the portability of LMDB across different frameworks has made it a go-to format, particularly for industries that use multiple frameworks for their artificial intelligence and DL efforts.

7.2.2 Other File I/O Frameworks

Various high-efficiency I/O frameworks have been developed for HPC. MPI-IO [156, 157] is a low-level parallel I/O library that provides generic unstructured data I/O support. HDF5[1] and NetCDF [37], on the other hand, provide high-level I/O libraries for structured scientific application data via feature-rich programming interfaces. The parallel variants of these libraries [58, 95] leverage MPI-IO to enable parallel access and storage for files. These technologies are complementary to our work. While we used POSIX I/O in our work, our approach is not limited to it and can easily adopt any of the mentioned parallel I/O models instead.

We note, however, that although in theory MPI collective I/O is supposed to internally perform optimizations that limit I/O randomization, this is not always true in practice. In most MPI-IO implementations today, collective I/O significantly lags in performance compared with POSIX I/O. In fact, in our experiments, the performance of MPI collective I/O was much worse than that of POSIX I/O. The performance of MPI independent-I/O was comparable to, but not as good as, POSIX I/O.

[1] https://support.hdfgroup.org/HDF5

We point out that other frameworks, such as RocksDB[2] and HDF5, also use tree-based structures and allow for highly efficient sequential access to the database. Although random database access is possible, it is not as efficient as sequential access because the database layout is not deterministic—the layout cannot be computed unless all data records are already laid out in the database (essentially the same problem as LMDB). Similarly, TFRecord (TensorFlow's native database format) allows only for sequential database access. The central issue here is that the data samples are not indexed in a way that are suitable for parallel I/O (i.e., indexing is based on keys, rather than a numerical ordering). Thus, the lessons learned in our work are applicable to the above mentioned other frameworks, too.

7.2.3 Storage Architecture

Some researchers have worked around the issue of I/O in DL by using cluster systems where each node has its own permanent storage [80, 186]. Thus, the input data can be fragmented and the corresponding fragment placed locally on each node, instead of on the global filesystem. While such workarounds are possible, they are not practical in several scenarios, such as those that require DL algorithms to be executed on large supercomputing systems. Most supercomputer systems tend to host their data on a shared global filesystem and do not equip each node with its own permanent storage. In fact, for such shared global filesystems, reading from a large number of smaller files has been shown to be significantly worse than reading from a single large file because of the additional metadata traffic that it generates [110].

Having said that, on-node storage (e.g., NVMe [88] and solid-state drives [77, 109]) are becoming common in large supercomputing systems. Some new-generation supercomputers, for example, Summit at Oak Ridge National Laboratory and Cori[3] at the National Energy

[2] https://rocksdb.org
[3] http://www.nersc.gov/users/computational-systems/cori/

Research Scientific Computing Center, are equipped with on-node permanent storage using these technologies. Such on-node storage, however, is accessible only when the job is allocated to a particular node and is wiped clean when the job terminates or when a new job is allocated. Thus, any data that needs to be persistently stored across jobs must be fetched from the global filesystem. Some systems utilize on-node storage technologies in the form of burst buffers, where data staging can be performed prior to the job start. However, we remind the readers of this work that datasets used for training are often very large and cannot be simply replicated on the on-node storage of each node. Thus, using burst buffers would mean that the training dataset needs to be segmented across the burst buffers available on each node. As discussed in the work, this is not an easy task and would require the application to have prior knowledge as to what parts of the file would be accessed by each node. Unfortunately, traditional I/O systems used in DL do not have this knowledge, at least not without some of the improvements proposed in this work such as the data provenance information. Having said that, one could imagine combining the proposed data provenance technique with burst buffer technology to predict what data goes on which node and perform the necessary I/O before the job starts, that is, while the job is waiting in the queue. This is a viable technique that we have not explored in this work.

7.2.4 Input Pipeline Optimizations

Recently, researchers have realized the importance of I/O in DL. Consequently, a number of input pipeline optimization techniques have been proposed [23, 86, 88, 180, 191], for example, data caching, computation and I/O overlapping (pipelining/prefetching), parallel data parsing, and in-memory data shuffling. While these approaches are certainly useful, we believe that they are orthogonal improvements. For example, techniques such as data caching assume that all the data can fit in the system's memory for multiple epochs. This approach

is useful for small datasets but is obviously not a feasible optimization for larger datasets. Techniques such as prefetching can hide the I/O cost behind that of the computation, but they benefit only those cases where the computation is more expensive than the I/O itself. For single-pass algorithms (approaches that compute on the data only once), I/O is often more expensive than the computation. In contrast, our work solves the root causes of various I/O problems. In any case, these other input pipeline optimizations can be applied in conjunction with our proposed approach to further improve performance.

7.3 Communication Optimizations

Network I/O has been identified as one of the significant bottlenecks in distributed DL systems. A large body of work has been performed to optimize communication performance (i.e., tensor transfer during parameter update or between subgraphs on different machines) in distributed training. We present the gradient compression techniques, gradient synchronization optimizations, and communication frameworks in the following subsections.

7.3.1 Gradient Compression

Data compression is a way to reduce the communication data size which can effectively mitigate the communication bottleneck. A lot of active research in DL focuses on gradient compression, which can be carried out in two ways: gradient quantization, and gradient sparsification. Gradient quantization is to represent a set of continuous values of a gradient with a set of finite values, while gradient sparsification leverages some threshold to transfer only informative gradients selectively. The most challenging aspect of gradient compression is to minimize the amount of data transfer while maintaining the quality of training in term

of inference accuracy.

Gradient quantization has been shown to improve training performance of distributed DL in the literature. One-bit SGD [140], an aggressive data quantization approach, achieved 10-fold speed up with negligible accuracy loss on speech DNNs on 20 distributed machines. QSGD [13] adopts the stochastic quantization method along with an efficient lossless data encoding to compress gradients. It demonstrated 1.8-fold improvement on ImageNet and ResNet152 training on 16 GPUs. TernGrad [169] proposed three-level gradient quantization, which significantly improved performance for various DNNs. Both QSGD and TernGrad guarantee convergence and accuracy of the training. DoReFa-Net [190] modified AlexNet to use low-bitwidth gradients to reduce communication bandwidth. The research showed that DoReFa-Net achieved the similar inference accuracy compared to the original 32-bit AlexNet.

Similarly, gradient sparsification has been demonstrated to be useful for data communication in parallel DL. Gradient sparsification was adopted in DNN training on a commodity GPU cloud [150] by using a constant threshold for each parameter. The approach had shown good training scalability up to 80 GPU instances with no loss in accuracy or convergence rate; however, a threshold selection is difficult. Dryden et al. [40] extensively modified the one-bit and threshold quantization approaches and achieved nearly 2-fold speedup in some cases. Aji and Heafield [9] observed that most of the gradient updates were not useful; therefore, they dropped 99% of the smallest updates. The initial results showed that this extreme gradient sparsification achieved up to 49% speedup on MNIST training on 4 GPUs. AdaComp [20], the adaptive residual gradient compression scheme, automatically tunes the compression rate based on local activity. With AdaComp, the compression ratio can be as high as 200x without any degradation in model accuracies. Deep Gradient Compression (DGC) [102] adopted several existing techniques, including momentum correction, local gradient clipping,

momentum factor masking, and warm-up scheme, to efficiently compress gradients. DGC achieved up to 600x compression ratio without losing accuracy.

7.3.2 Gradient Synchronization Optimizations

The allreduce communication has been well studied in much literature [12, 55, 74, 176, 180]. Overlapping communication with computation is one of the most popular techniques. Our work currently does not include computation and communication overlap in our optimization, but we plan to revisit this optimization in the future. The similar approach to Horovod-TOPO has been proposed in [74]. Importantly, though, it does not directly handle tensor ordering—in fact, it ignores tensor dependencies and forces tensor transfers to be in the reverse order of the layers (i.e., assuming that tensor transfers only happen in the backpropagation phase). We believe that this approach is not applicable to many graph structures. For example, such method would not work with the synchronization of batch-normalization statistics in [180] that takes place in both forward and backward passes and the statistics have dependencies between each other. In contrast, our work analyzes the actual dependency structure in the graph and creates a tensor ordering schedule that would work for any static graph. We note that our work tackles the computational imbalance problem, rather than allreduce directly, which is different from the related work.

7.3.3 Communication Frameworks

Optimizing network I/O via data compression requires algorithmic changes in the DL training. In the case that adjusting the learning algorithm is not preferable, we can optimize the network I/O performance by improving the communication software.

Typically, TCP is used as a default communication protocol in most state-of-the-art DL

frameworks. For instance, the most popular DL framework, TensorFlow, uses gRPC [1, 2, 24] as the backbone communication protocol for its distributed training. gRPC internally uses HTTP/2 [54] which operates on top of TCP. However, gRPC/TCP is not a universal communication library option for supercomputing systems because it either is not supported or performs poorly on most supercomputers.

On the other hand, MPI [4] is more supercomputer friendly because it has been well optimized for a broad range of communication protocols for high-performance computing systems. MPI collective communication provides highly optimized group communication functionalities among a large number of processes and nodes [159]. Several DL frameworks, including TensorFlow, Caffe2, and CNTK, adopt MPI as a native communication protocol.

Because GPUs are becoming the most popular accelerators for DL, various GPU-aware MPI implementations also exist [10, 11, 14, 72, 73, 108, 151]. In addition, the GPU-specific data movement optimizations, such as the CUDA unified memory technology [100] and communication-computation overlap techniques [32, 33], have been proposed to improve the performance of data transfer between host and device

Verbs [106] is a network I/O library for interfacing with the InfiniBand architecture. With Verbs, applications can utilize the Remote Direct Memory Access (RDMA) capability, that allows a machine to access data in a memory of another machine over the network by by-passing the processing of CPUs on that system, provided by the InfiniBand hardware. Similar to MPI, various DL libraries [155, 179] have natively support Verbs for communication.

Several recent communication middlewares for DL are implemented on top of MPI or Verbs. Gloo [43], Facebook's collective communication library for Caffe2, has an implementation on both TCP and Verbs. Note that Gloo uses MPI for machine rendezvous, but not for underlying communication. However, Gloo has implemented its collective operations based

on the designs of MPI collective algorithms [159]. The Cray Programming Environment (CPE) Machine Learning (ML) Plugin [109] is built on top of MPI. Despite its impressive scaling, however, CPE ML's operations are blocking synchronous (as opposed to Horovod's nonblocking asynchronous operations), making them susceptible to deadlock, and are invalid in some computation graphs. Aluminum [39], an asynchronous GPU-aware communication library, contains a novel latency-optimized Allreduce algorithm to improve the performance of communication that overlaps with computation.

Baidu utilizes MPI point-to-point operations to compose its own allreduce library, namely `baidu-allreduce` [134] based on a novel ring-algorithm [50, 126]. Uber addresses inefficiency of the default tensor transfer of TensorFlow by proposing Horovod. Horovod is developed based on `baidu-allreduce` by replacing the ring algorithm with high-performance collective communication APIs including MPI, NVIDIA NCCL, IBM DDL, Facebook Gloo, and Intel MLSL. Moreover, Horovod proposes an optimization approach, called "tensor fusion", to coalesce small tensors into one buffer before performing the actual data transfer to reduce the data transfer overhead. Horovod has shown impressive performance in the literature [88, 142]. However, its loss of scalability on large-scale systems is well documented [173]. Kurth et al. [87] utilized Horovod-MLSL (Intel Machine Learning Scaling Library backend to Horovod) and reported that it required additional thread binding to avoid the background threads from monopolizing CPU cores.

Overall, to the best of our knowledge, our work on computational imbalance is the first to identify the contention between data-processing components as a cause of TensorFlow/Horovod's scalability limitation.

7.4 Algorithmic Improvements to Parallel Deep Learning

Another crucial DL research area involves high-accuracy large-batch training. Using large batches of data samples to train DNNs on large-scale supercomputers is a common practice for achieving high parallelism. In doing so, however, the inference accuracy can degrade significantly since the DNN parameters are updated less frequently with gradients that contain more information. Consequently, several ongoing studies have been trying to improve the inference accuracy of large-batch training.

The common key idea of these techniques is to adjust the "learning rate" or LR which controls the magnitude of parameter changes in the DNN training. One of the earliest approaches in this direction involves adjusting the global LR linearly [82] based on the size of the batch. For instance, if the batch size is scaled by k times, the LR is also scaled by k. This approach is risky, however, and can cause the training to diverge during the initial phase. To address this issue, a warm-up scheme was introduced in [55] to prevent such divergence by starting with a small LR and increasing it later during the training. Various SGD-based LR adjustment schemes have been proposed and have significantly improved the robustness of SGD while using large batch sizes, for example, Adagrad [41] dynamically adjusts LR based on the gradient magnitudes, Adadelta [185] solves the shortcomings of Adagrad by applying the squared gradient exponential decay method (similar to using momentum [52]) to reduce its sensitivity to the initial global LR, and Adam [81] adds the gradient exponential decay to address flat minima in the error surface. You et al. [181, 183] proposed layer-wise adaptive rate scaling (LARS). LARS uses a different LR for different layers in the DNN, where the LR of a layer is the ratio between the norm of the layer weights and the norm of the gradients. With these optimizations, LARS successfully enables parallel training using

large batch sizes—up to 32,768—with negligible loss in inference accuracy.

At the time of writing this thesis, the largest batch size known is 131,072 [122]. These studies demonstrate that training with large batch sizes is practical and needs to be optimized, a subject that is the target for this work.

Chapter 8

Summary and Discussion

We present a summary of the work performed as a part of this thesis in Chapter 8.1. We list the peer-reviewed publications related to this thesis work in Chapter 8.2. Discussions on some of the lessons learned during this thesis work and on some open questions on large-scale DL are presented in Chapter 8.3. Finally, we provide some ideas on tradeoffs between performance and accuracy that can be extended in the future in Chapter 8.4.

8.1 Thesis Summary

This thesis addresses two significant scalability limitations in large-scale DL, namely data movement from the filesystem (i.e., file I/O) and computational imbalance in data processing.

For the first limitation, we showcased the file I/O bottleneck through the analysis and optimization of Caffe/LMDB on up to 9,216 cores of the Bebop cluster. On such large-scale system, we observed that up to 90% of the overall DL training time was devoted to data loading from a shared filesystem. We thoroughly analyzed LMDB and found that the main causes of its shortcomings were from the use of `mmap` and its nondeterministic database layout. These problems are quite common among existing file I/O subsystems for DL. We proposed LMDBIO—an optimized I/O plugin for scalable DL. A summary of the six optimizations that comprise LMDBIO is shown in Table 8.1. All LMDBIO optimizations outperform LMDB in all cases and improve overall application performance by up to 65-fold

in some cases. In fact, on our system, these optimizations can saturate the system's available I/O bandwidth for DL frameworks.

Table 8.1: LMDBIO optimization summary

Library	Optimization	Reducing Interprocess Contention	Using Explicit I/O	Eliminating Sequential Seek	Managing I/O Size	Reducing I/O Randomization
LMDB	-					
LMDBIO	LMM[1]	✓				
	LMM-DM[2]	✓		(partial)		
	LMM-DIO[3]	✓	✓			
	LMM-DIO-PROV[3]	✓	✓	✓		
	LMM-DIO-PROV-COAL[3]	✓	✓	✓	✓	
	LMM-DIO-PROV-COAL-STAG[3]	✓	✓	✓	✓	✓

[1] Intra-node file I/O optimization.
[2] Speculative distributed file I/O optimization.
[3] Direct file I/O optimization.

For the second limitation, we demonstrated the impact of computational imbalance in the processing of large-scale DL via the analysis and optimizations of TensorFlow/Horovod on the Summit supercomputer. Modern DL frameworks, including TensorFlow, allow multiple data-processing components to operate on the same hardware asynchronously at the same time causing them to compete with each other for resources and lead to computational imbalance. We proposed four optimizations to overcome its shortcomings. The summary of our optimizations is shown in Table 8.2. With our optimizations, we are able to improve the performance of the training of several real-world DNNs by 35% in some cases on 24,576 GPUs of Summit—the world's fastest supercomputer today.

The scalability problems that we addressed in this thesis are only a subset of a broad range of issues that have been identified in large-scale DL systems. Achieving perfect scalability

Table 8.2: Computation imbalance optimization summary

Library	Optimization	Eliminating `MPI_Allreduce` Out-of-Sync	Eliminating Resource Competition
Original Horovod	-		
Our Horovod	GS	(partial)	(partial)
	NBCS		(partial; better than GS)
	SCP		✓
	TOPO	✓	✓

as well as a good accuracy is still a hard problem in DL. Although we have not solved all the problems that we have identified, we provide some thoughts on ways to improve the scalability of DL in the subsequent sections.

8.2 List of Publications

The following is the list of peer-reviewed publications related to this thesis.

Chapter 3: Intra-node File I/O Optimization

- Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Towards Scalable Deep Learning via I/O Analysis and Optimization. In *Proceedings of the 19th IEEE International Conference on High Performance Computing and Communications (HPCC)*, 2017, pp. 223-230. [131]

Chapter 4: Inter-node File I/O Optimization via Speculative Parallel I/O

- Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Parallel I/O Optimizations for Scalable Deep Learning. In *Proceedings of the 23rd IEEE International Conference*

on Parallel and Distributed Systems (ICPADS), 2017, pp. 720-729. [130]

- Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Parallel I/O Optimizations for Scalable Deep Learning. (Poster.) *EuroMPI/USA*, 2017.

Chapter 5: Direct File I/O Optimizations

- Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Scalable Deep Learning via I/O Analysis and Optimization. In *ACM Transactions on Parallel Computing (TOPC)*, 2019, 6, 2, Article 6 (July 2019), 34 pages. [132]

- Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. File I/O Optimizations for Large Scale Deep Learning. (Poster.) *The 7th Annual MVAPICH User Group (MUG) Meeting*, 2019.

- Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. I/O Bottleneck Investigation in Deep Learning Systems. (Poster.) *The 47th International Conference on Parallel Processing (ICPP)*, 2018. **Best Student Poster (Best Ph.D. Forum) Award**.

Chapter 6: Computation Imbalance Optimizations for Data Processing

- (Under review.) Sarunya Pumma, Daniele Buono, Fabio Checconi, Xinyu Que, and Wu-chun Feng. Alleviating Load Imbalance in Data Processing for Large-Scale Deep Learning. *The 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020. [128]

- Sarunya Pumma, Daniele Buono, Fabio Checconi, Xinyu Que, and Wu-chun Feng. Optimizing Large-Scale Deep Learning by Minimizing Resource Contention for Data Processing. (Poster.) *The 2nd IBM IEEE CAS and EDS – AI Compute Symposium*, 2019. **Best Student Poster Award**.

Other publications

The following publications are not directly related to this thesis. However, they are related to the preliminary work that I studied at the beginning of my Ph.D. prior to this thesis.

- Hao Wang, Jing Zhang, Da Zhang, Sarunya Pumma, and Wu-chun Feng. PaPar: A Parallel Data Partitioning Framework for Big Data Applications. In *Proceedings of the 32nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017, pp. 605-614. [168]

- Sarunya Pumma, Wu-chun Feng, Phond Phunchongharn, Sylvain Chapeland, and Tiranee Achalakul. A Runtime Estimation Framework for ALICE. *Future Generation Computer Systems*, 72:65–77, 2017. [129]

8.3 Discussion

In this section, we provide the discussion related to this thesis work. The discussion aims to address some open problems in large-scale deep learning.

8.3.1 What Would the Ideal Filesystem for Deep Learning Look Like?

While our file I/O study provides an empirical evaluation of some of the I/O problems in large-scale DL and some solutions to these problems, we would like to take a moment to discuss the broader lessons that we learned from this study. One important takeaway is that several of the solutions proposed in our work are effectively workarounds for problems in the filesystem. A more comprehensive and elegant solution instead would be to improve

or develop a new filesystem that is more targeted to DL workloads. What would such a filesystem look like? We have some thoughts.

1. Deep learning workloads are read-heavy and rarely ever do writes. In fact, most DL frameworks perform writes only for checkpointing purposes, and these writes happen to files that are disjoint from the database file. In other words, the database files are "read only" for the lifetime of the application, and the checkpoint files are "write only" for the lifetime of the application. If these files are separated onto two different filesystems, each filesystem can be modified to support much more restrictive semantics. For example, the read-only filesystem can perform aggressive caching of global data on local nodes and avoid any locking and state management overheads needed for such data consistency. Similarly, the write-only filesystem does not have to concern about data consistency (the writes are nonoverlapping) and need not perform any caching at all.

2. The ideal filesystem for DL would be one that supports fast random access similar to main memory. Thus the random data batch composition requirement of the training algorithms, namely, SGD, can be satisfied through data reading, and the additional in-memory data shuffling can be completely avoided. Technologies such as on-node non-volatile random-access memory or NVRAM and consortia such as Gen-Z[1] are already working in this direction, so such an approach might not be completely off the table. We note, however, that practically using such technologies is still some time away at the time of writing this thesis and avoiding random access is perhaps still the best strategy for now.

3. If random access is impractical for filesystems, the next best option would be strided access. Strided accesses are, unfortunately, not well supported by filesystems. I/O

[1] https://en.wikipedia.org/wiki/Gen-Z

access in DL is very structured and is regularly strided. Moreover, there are no "holes" in the data access. All bytes are accessed by one process or another. Filesystems typically do not provide native APIs for such access, thus resulting in unnecessary prefetching and cache flushing. We worked around this problem with our staggered I/O model, but that model serializes I/O, which could have been entirely avoided if the filesystem had provided better strided I/O access.

8.3.2 Rethinking Process/thread Synchronization in DL Communication Subsystems

Process/thread synchronization is critical for performance in multiprocessor programming. There exist two main classes of process/thread synchronization approaches [65]: *spin waiting* and *blocking*, each of which is suitable for the different performance requirements. Spin waiting or busy waiting is an approach that a process/thread repeatedly checks whether an expected event has happened, while blocking or yielding is a method where a process/thread suspends itself to allow the OS to schedule another process/thread to run while it is waiting for some event to occur. Spin waiting consumes CPU cycles, thus it is suitable for the case where the arrival time of the expected event is small. In contrast, blocking is appropriate for the case that the arrival time of the event is long and larger than the context switching time. In this section, we will discuss the process/thread synchronization in the context of network I/O subsystems in DL.

As mentioned in Chapter 7.3.3, various DL frameworks and high-level communication frameworks, including Horovod, are built on top of MPI. The pros and cons between the spin waiting and blocking approaches for checking incoming messages have been long studied by MPI researchers and developers. The spin wait method has been proven to be better for

latency compared to the blocking approach. For this reason, all MPI implementations today adopt spin waiting within their blocking communication operations. MPI is not designed to support applications that oversubscribe CPU cores.[2] It assumes that there are sufficient cores to run MPI processes/threads. Unfortunately, that is not the case for modern DL frameworks. They usually oversubscribe cores via multithreading. As a consequence, spin waiting within MPI can cause performance degradation in parallel DL as presented in Chapter 6. In our work, Horovod-NBCS is the workaround of the spin wait problem—it uses nonblocking MPI operations and manually and periodically yields the Horovod background thread. However, the best way to tackle this problem is to avoid using MPI when the core oversubscription occurs.

Given that the core oversubscription is present, the gRPC's model for checking for new messages might be a more suitable for DL. gRPC uses a combination of spin waiting and blocking—that is a gRPC thread/process loops over the `epoll_wait` call with timeout until a new message arrives. `epoll_wait` is a blocking system call that puts the caller process/thread to sleep until the awaited event shows up or timeout expires. With this model, the gRPC process/thread gives up the core occasionally for others. Although the gRPC process/thread synchronization model sounds appealing, its performance is questionable (as discussed in Chapter 7.3.3).

Since there is no-one-size-fits-all communication subsystem for DL while the core oversubscription occurs, we can either reimplement a communication method that nicely balances between spin waiting and blocking or directly solve the core oversubscription problem (we present a way to solve this problem in Chapter 8.3.3).

[2]Some MPI implementations have an option to yield a process/thread when it is idle waiting, for example, OpenMPI provides the `mpi_yield_when_idle` knob to control the synchronization behavior (https://www.open-mpi.org/faq/?category=running#oversubscribing).

8.3.3 Enhancing Intra-node Parallelism of DL Frameworks via Lightweight User-Level Threading Libraries

As mentioned in Chapter 6.1, there are multiple data-processing components running in the DL environment. Typically, an on-node parallelism on CPUs is provided via multiple threads. Modern DL frameworks, e.g., TensorFlow and PyTorch, employ a similar threading model—that is each component is associated with at least one *thread pool*, which can be either private or shared with other components. Each pool contains one or more threads which are normally the OS-level threads, i.e., **Pthreads**, which their management cost is known to be expensive. These threads are forked when the pool is being created and joined once the pool is being destructed. Each pool has a task queue where each enqueued task is picked up and run by one thread in the pool until the task is completed. When the task queue is empty, all threads sleep. In common practice, each thread pool manages its own threads *separately* from other pools in the system—which is a coarse-grained management of threads. Consequently, users have to carefully configure a number of threads in each pool to avoid CPU oversubscription which can degrade the entire execution performance.

Despite threads within a DL framework, some external libraries/plugins have their own threads—for example, multithreaded Intel MKL is based on OpenMP [34], and Horovod has one background thread for handling data transfer requests and one thread for each data transfer to check for completion of some collective communication APIs (e.g., NVIDIA NCCL). These libraries/plugins mostly rely on OS-level threads. To avoid oversubscription of OS-level threads, users have to rigorously tune the amount of threads used in each library/plugin, which it is a nontrivial task. Figure 8.1 illustrates the oversubscription of OS-level threads example. In the figure, more than one thread is running on each core.

In Chapter 6, we demonstrated that although we have carefully configure the number of

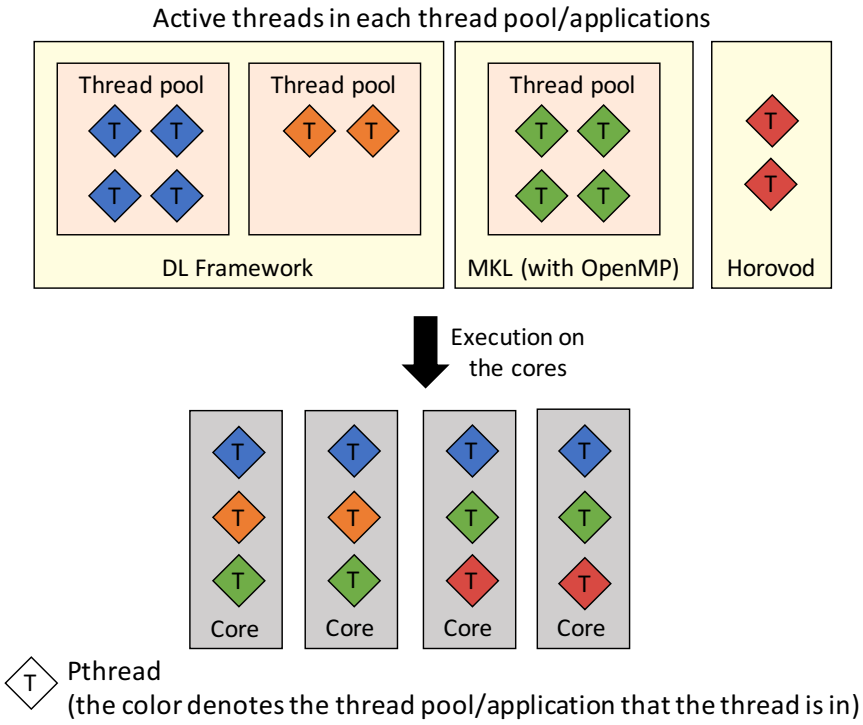

Figure 8.1: Example of oversubscription of OS-level threads

threads in the DL framework and external libraries/plugins, oversubscription of OS-level threads still occurs causing threads to oversleep which leads to computational imbalance. To actually solve the core oversubscription problem, the heavyweight OS-level threading approach can be replaced with *lightweight user-level threading* (LULT) library, such as Argobots [141] and BOLT [70]. The main advantage of the LULT library over the OS-level one is that its thread management is *less* expensive.

The LULT library provides an abstraction over the OS-level thread library in order to reduce the thread synchronization cost and alleviate the core oversubscription problem. The execution model of LULT is shown in Figure 8.2. In case of Argobots, it provides execution streams and lightweight work units (i.e., a user-level thread or a tasklet; although there are many types of work units, we will loosely call each work unit "lightweight thread"). Each execution stream is mapped to an OS thread, and it has one associated scheduler which is

responsible for scheduling lightweight threads to run in the stream. As long as the number of execution streams is not more than the number of cores, the user applications—DL framework, MKL, and Horovod—can create as many lightweight threads as they would like without the risk of oversubscribing the cores.

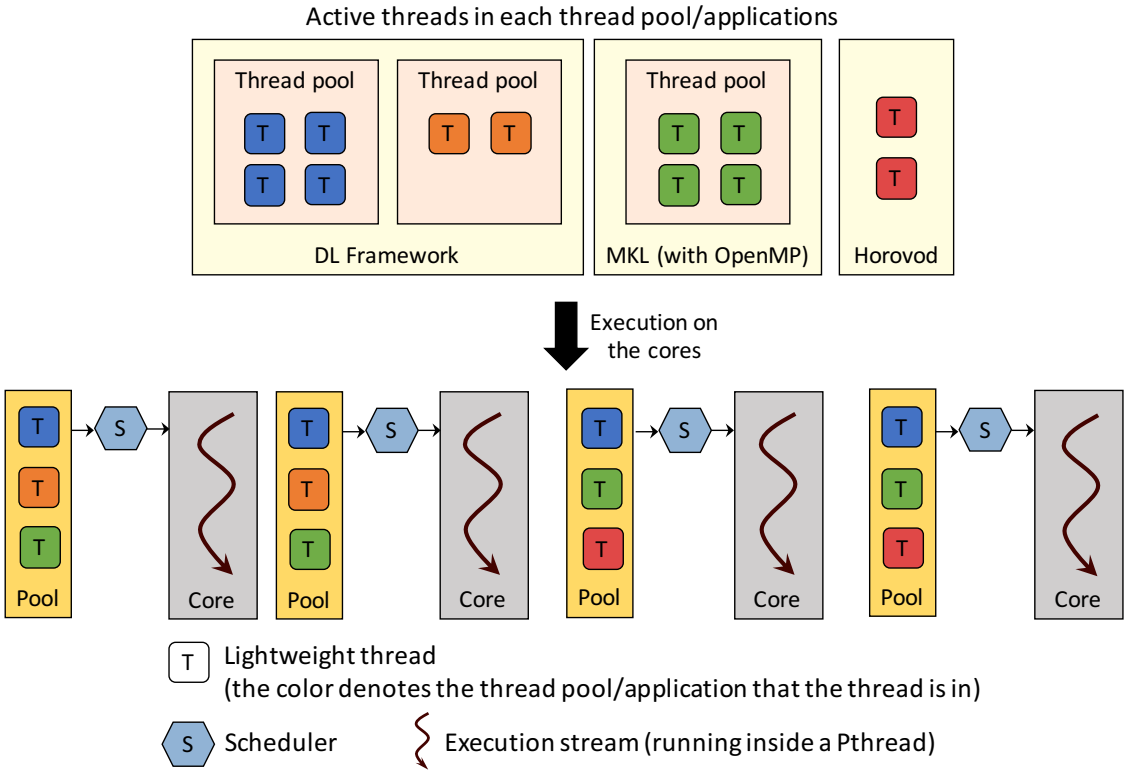

Figure 8.2: Example of using lightweight user-level threading library in DL

We have not performed an in-depth study on the use of LULT library in the DL execution environment. However, we believe that it is an alternative approach to avoid computational imbalance presented in Chapter 6 as explained above.

8.3.4 Can We Beat the ImageNet-ResNet50 Training World Record?

Since 2017 [55], there has been ongoing informal "fastest time to convergence of the ImageNet-ResNet50 training" competition in the HPC community. The training must converge to the state-of-art top-1 testing accuracy of 75% or more. Several researchers and companies have been improving the training algorithm and strategy in order to beat the world record. The world record currently belongs to Fujitsu [176] with a training time of 74.7 seconds and top-1 accuracy of 75.08%.

Our work utilized Summit, the world's fastest supercomputer at the time of this writing, but this supercomputer still falls short of the world record. While this thesis does not directly target beating the world record, we thought that it would be interesting to identify why Summit falls short of this goal. Figure 8.3 compares our image processing rate with that of the current world record. Our best training performance in term of image processing rate is 0.35 millions images per second, while the one of the world record is 1.73 millions images per second. Our performance is *five* times worse than the world record even though Summit is approximately 5.4 times faster than the AI Bridging Cloud Infrastructure (ABCI) cluster that Fujitsu used (Summit's and ABCI's peak performance are 200 petaflops and 37 petaflops, respectively). This suggests that Fujitsu is able to utilize its cluster more efficiently than Summit.

In this section, we present **six observations** on the shortcomings of our work and the advantages of Fujitsu's work:

Observation 1: Global batch size (GBS)—*We cannot achieve state-of-art accuracy using a large GBS.* The largest GBS that we can use without accuracy loss is 24,576 which is very close to the best known GBS prior to Fujitsu's work (that is, GBS of 32,768 [181, 182]). However, Fujitsu has claimed that they are able to maintain the accuracy while using 3.3

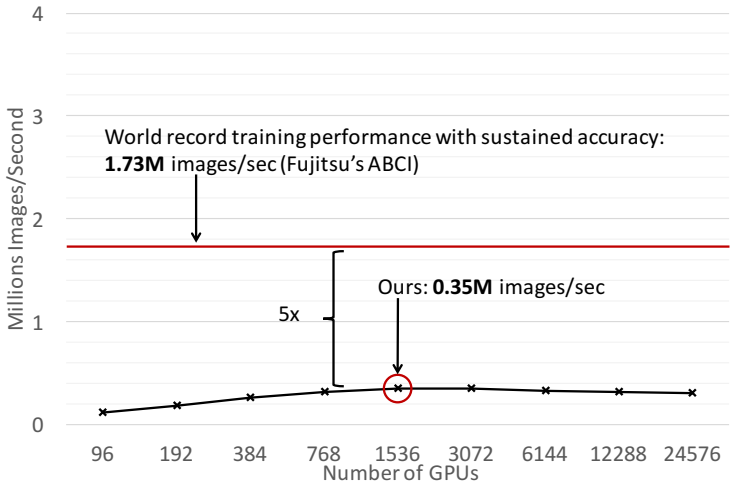

Figure 8.3: Our ImageNet training performance vs. the world record (the graph is showing strong scaling—the state-of-art top-1 accuracy is maintained)

times larger GBS, that is 81,920. With such large GBS, Fujitsu has 3.3 times more data parallelism compared to ours. However, we are not able to reproduce their claim.

Solution 1: We need to be able to reproduce Fujitsu's claim to achieve the same level of parallelism *or* use a larger GBS than Fujitsu's (while sustaining accuracy) to achieve more parallelism.

Observation 2: CPU/GPU architecture—*Summit is more GPU biased than ABCI.* Summit has smaller CPUs-to-GPUs ratio than ABCI (7 CPU cores per GPU for Summit and 10 CPU cores per GPU for ABCI), which makes it harder for Summit to dispatch work to the GPU fast enough. As a consequence, the GPUs on Summit are more susceptible to be idle than those of ABCI.

Solution 2: *Hardware improvement*—Adding more CPU cores to the system is one way to address this problem. The additional CPU cores can help improving the performance of GPU kernel dispatching.

Software improvement—We can offload some tasks from the CPUs to run on the GPUs, e.g.,

data augmentation in the input pipeline, to free the CPU cores for task dispatching.

Observation 3: Network architecture—*Communication software is the bottleneck.* Traditional network hardware requires large buffers to achieve the peak performance. Thus, communication software, such as, Horovod, fuses data buffers to create large buffers, although it adds overhead to data transfers. However, current network hardware, including InfiniBand on Summit, allow for efficient communication even with small message sizes (i.e., a few kilobytes of data).

Moreover, most current network hardware provides multiple concurrent communication contexts, but the access to them is serialized in the software, for example, Horovod has one thread per process to handle concurrent data transfers.

Solution 3: If the communication software can directly access the network hardware or through a lightweight software stack, e.g., Mellanox Scalable Hierarchical Aggregation and Reduction Protocol or SHARP [56], the buffer fusion step can be bypassed. Moreover, if multiple network hardware contexts can be exposed to the software, we can inject simultaneous data transfers using multiple threads to increase the data transfer parallelism.

Observation 4: GPU computation—*Computation kernels are not always GPU optimized.* Although kernel fusion can help improve the performance of the GPU kernel execution, the current method only fuses consecutive kernels. However, Fujitsu hand-optimized their kernels by fusing nonconsecutive kernels to increase the GPU utilization. In addition, the current graph compiler used in our work, XLA, disables the communication-computation overlap as explained in Chapter 6.2.1.

Solution 4: XLA should be improved to fuse nonconsecutive kernels as well as provide communication and computation overlap.

Observation 5: OS scheduler—*Modern Linux versions use the completely fair scheduler (CFS) which is not always fair.* CFS only schedules threads/processes based on the CPU usage time, which can be a problem for asynchronous event based applications, including GPU applications. For instance, event polling threads, e.g., MPI threads, might not be able to yield when they need. Because their CPU usage times are small, CFS will reschedule them quickly once they yield causing them to occupy CPU cores for longer than what they actually need. This can potentially prevent other threads from using the cores.

Solution 5: OS scheduler should take into account other information in scheduling, such as a type of a process/thread.

Observation 6: DL framework (minor observation)—*We use TensorFlow which might not be the best DL framework.* There exist several open-source DL frameworks as presented in Chapter 7.1. While TensorFlow was adopted by a few previous world record holders [74, 180], majority of the world record's work did not use TensorFlow. For instance, Caffe and Caffe2 were utilized in the early period of the competition [55, 63, 181, 182]. Later on, other frameworks were used and able to break the world record [12, 113]. Recently, Fujitsu used MXNet.

Solution 6: An empirical evaluation of DL frameworks is needed to determine which DL framework is the best on Summit.

Based on the solutions presented in the observations above, we should be able to close the performance gap between our work and Fujitsu's. Perhaps, we might be able to beat the world record in the future.

On a related note, MLPerf [111],[3] a benchmark suite for machine learning, was released in

[3] https://mlperf.org

February 2018 to establish fairness in benchmarking machine learning software and hardware. MLPerf includes the ImageNet-ResNet50 benchmark. However, the world record work mentioned earlier in this section did not use MLPerf. At the time of writing this thesis, the latest MLPerf is v0.6. Based on the MLPerf Training v0.6 results,[4] Google [85] achieved the best ImageNet-ResNet50[5] training performance on 2,048 TPU-v3 cores using the GBS of 32K with the training time of 76.8 seconds and the top-1 testing accuracy of 76.9%.

8.3.5 Compatibility of Our Work to Modern DL Frameworks

This thesis adopts two of the most popular DL frameworks in their times, Caffe and TensorFlow. However, as the DL domain is evolving very quickly, other DL frameworks that contain more recent features and capacities might become more popular in the future. For instance, at the time of writing this thesis, Caffe is already outdated. From the growth score shown in Figure 8.4, TensorFlow is still the most popular DL framework, and PyTorch is gaining much popularity which it might overtake TensorFlow's place in the future. To address this issue, we discuss how the approaches presented in this thesis can be used in the modern DL frameworks, specifically PyTorch, in this section.

One of the main differences between TensorFlow and PyTorch is their programming styles. TensorFlow adopts the *declarative* paradigm that is a computation graph is defined prior to the execution takes place, whereas PyTorch inherits the *imperative* model from Python [125] where the structure of a computation graph is not predefined but realized when the graph is being executed. As TensorFlow is a declarative-style DL framework, static computation graphs, which the graph structures remain unchanged throughout the execution, are inherently supported. However, a limited dynamic computation graph support has been recently

[4] https://mlperf.org/training-results-0-6/

[5] Quality target: 75.9% top-1 accuracy; dataset: ImageNet (224 x 224); reference implementation model: ResNet50 v1.5

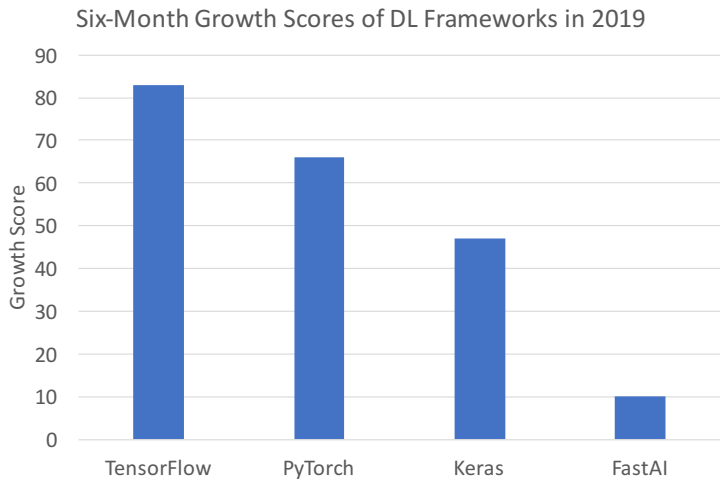

Figure 8.4: Six-month growth scores of DL frameworks in 2019 [59]: the growth scores are calculated based on six criteria, i.e., Google search interest, GitHub activity, Quora followers, Medium articles, ArXiv articles, and online job listings

added to TensorFlow v.2.0. PyTorch supports both static and dynamic graphs, but due to its imperative nature, the graphs are dynamically created and freed at runtime.

However, PyTorch is made up of various data-processing components similar to those of TensorFlow (see Chapter 6.1 for more details). Each component provides an external library/plugin support. Thus, our LMDBIO and optimized Horovod can be used as file I/O and network I/O plugins for PyTorch. Below, we will discuss some issues that users have to be aware of when adopting our work in PyTorch.

LMDBIO-PyTorch interfacing overhead: PyTorch is a C++ & Python based DL framework. Its backend is mainly written in C++. However, the Python backend is also heavily optimized for performance and seamless integration with Python—which is used as the main frontend. PyTorch supports the file I/O subsystem plugin via the Python interface, namely `Dataset`[6]. LMDBIO has the Python API support. However, using LMDBIO via the Python interface incurs an overhead in transforming the read data from the C++ data buffer to

[6] https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset

the `Tensor`[7] PyTorch's Python object. We have not studied this overhead in depth, but the addition cost can be significant for large datasets. The better way to use LMDBIO in PyTorch is to connect them in the C++ level to avoid the C++-Python interfacing overhead. However, doing so requires an intrusive modification of the PyTorch code. We note that LMDBIO should be able to be used via the C++ APIs without any modification to the PyTorch code as well as additional interfacing overhead.

Static graph limitation of Horovod-TOPO: Our first three optimized Horovods, Horovod-GS, Horovod-NBCS, and Horovod-SCP, support dynamic computation graphs as they inherit the workflow of the original Horovod background thread. As Horovod-TOPO assumes that the dependencies of the tensors to be transferred in the same graph are constant throughout the execution, it supports only static computation graphs. To leverage Horovod-TOPO for static graphs in PyTorch, Horovod-TOPO has to compute tensor dependencies during the first execution of the graph. Then, it can use such information to compute the tensor ordering and fusion schemes once the first iteration execution is finished. Although Horovod-TOPO is currently restricted to static computation graphs, we believe that we can extend the dynamic graph computation capability to Horovod-TOPO with a small engineering effort.

Despite the limitations discussed above, we believe that our LMDBIO and optimized Horovod will able to improve the scalability performance of PyTorch as well as other modern DL frameworks

[7] https://pytorch.org/docs/stable/tensors.html

8.3.6 Tradeoff Between Batch Size, Convergence Period and Accuracy

In the large-scale DL domain, using a large batch size is a popular technique to increase the training parallelism. However, large batch sizes tend to decrease the inference accuracy due to a small number of parameter updates. To address such problem, a broad range of approaches have been proposed. One of the most popular techniques is to train for a larger number of iterations to allow the parameters to be updated for more times. In some cases, the training can converge to a desire inference accuracy within a reasonable amount of additional training iterations. In other cases, the additional training iterations becomes too large making the use of the large batch size not worth it.

In this section, we discuss the tradeoff between batch size, convergence period—in term of a total number of epochs to reach the desire accuracy, and inference accuracy. We demonstrate our discussion through the CIFAR10-AlexNet training[8] example. We tune the hyperparameters via an empirical study. We note that the different batch sizes have the different hyperparameter settings[9]. We set the target accuracy to 0.85 and the maximum training epochs to 2000. The training is considered converged if a moving average of the last five epochs reaches the target accuracy. If the training cannot reach the target inference accuracy within 2000 epochs, we consider it not converged.

Our criteria in choosing the batch size that gives the best tradeoff between the three factors mentioned above is that by doubling the batch size, we gain two times more parallelism. Thus, if we double the number of processors for the training, as long as the number of

[8]The experiments are run on the Theta cluster at Argonne National Laboratory (https://www.alcf.anl.gov/theta) using a single processor.

[9]Hyperparameters for batch sizes 32 - 2048: SGD with momentum = 0.9, linear learning rate scaling with maximum initial learning rate = 5.12, and linear warm-up scheme. Hyperparameters for batch sizes 4096 - 16384: SGD with LARS, linear learning rate scaling with maximum initial learning rate = 5.12, and linear warm-up scheme.

Table 8.3: Batch size vs. convergence period (a total number of epochs to reach the target inference accuracy) of the CIFAR10-AlexNet training. The target inference accuracy is 0.85 and the maximum training epochs is 2000.

Batch Size	Convergence Period (Epochs)
32	104
64	103
128	103
256	104
512	104
1024	117
2048	144
4096	1198
8192	1483
16384	Does not converge

epochs does not increase by more than double, we will still get some benefits from using the larger batch size. Therefore, the best batch size is 2048 in our example. We note that it is not worth it to jump from the batch size of 2048 to 4096 as the convergence period increases by ∼8 times. Therefore, in the data parallel execution of the CIFAR10-AlexNet training, we can use at most 2048 processors. In the perfect scenario, we will be able to improve the training time by 64 times using the batch size of 2048 compared to the training using the batch size of 32 (given that the convergence periods for the two batch sizes are almost the same).

8.4 Future Work: Tradeoffs Between Data Movement and Accuracy

In this thesis, we focus mainly on the scalability performance of deep learning in large scale. We have touched upon the accuracy aspect of DL only slightly in Chapters 6 and 8.3.6.

However, both scalability and accuracy are important to the large-scale DL execution. They tend to have a definite duality between each other. Improving one tends to hurt the other. Thus, as a future work, we propose to explore the *tradeoffs* between training performance and inference accuracy. There are several aspects that we can study in this area. We present three potential studies to give a general indication of the kind of optimizations possible in this area. Two of these aspects address the data movement and inference accuracy tradeoffs, while the third addresses process synchronization overheads, which can loosely be considered to be a network I/O problem.

8.4.1 Data Reuse Optimization

In general, the training process of DL iterates through samples in the dataset multiple times to improve the trained model quality until it reaches satisfied prediction accuracy. This training scheme is referred to as "multi-pass training". In common practice, each data sample is visited once every epoch (i.e., all data samples in the dataset). Therefore, the state-of-art "reuse distance" is equal to one epoch. As the datasets nowadays are in the scales of hundreds of terabytes and petabytes, the reuse distance can be very large. Large reuse distance has a positive impact on the inference accuracy; however, it hurts data reuse as the data can already be taken out of memory by the time that it is revisited.

Since data samples are trained several times, it is essential to perform "data shuffling" to add some variance into the input batches. The convergence rate of the training highly depends on the degree of randomness in which group of samples is processed in each iteration [16]. The ideal data shuffling is to perform it among all the samples in the entire dataset, but because the total amount of memory in the system is usually smaller than the dataset size, the memory cannot accommodate the entire epoch of data. Therefore, in common practice,

data shuffling is performed within a few batches of data that are in memory instead of across the whole epoch.

The goal of this proposed work is to investigate tradeoff between reuse distance and data movement. The core idea of this work is to reuse the data samples while they are still in memory. In other words, our approach attempts to minimize the data movement, both from filesystem to memory and in the network I/O, to improve training performance and to maximize the reuse distance to maintain inference accuracy.

Table 8.4: Tradeoff between reuse distance and data movement

Level	Max In-Memory Data Size
Intra-process	M
Per node	M × N
Per switch	M × N × S
Per rack	M × N × R
Per supercomputer	M × N × SC

M = amount of memory per process
N = a total number of processes per node
S = a total number of nodes per switch
R = a total number of nodes per rack
SC = a total number of nodes in the entire supercomputer

Our approach exploits the benefit of a large-scale system to use multiple process memories collaboratively to maximize the reuse distance. To minimize data movement, we use the topology-aware subbatch shuffling approach. Note that a partition of a batch of input is called "subbatch", where the size of a subbatch is equal to B/P (B is a batch size, and P is a total number of processes). Table 8.4 presents multiple levels of subbatch shuffling.

From Table 8.4, the way to compose a batch of input with the least data movement is to perform subbatch shuffling within processes. In this case, although data movement is minimal, a variety of data and reuse distance are also very small which can hurt the inference

accuracy. In order to increase the reuse distance, subbatch shuffling can be carried out between processes on the same node or the same switch or the same rack or the same supercomputer. The farther the data shuffling is performed, the higher the reuse distance and the better the variety of the data, however; more data movement is required. Thus, a balance between reuse distance and data movement has to be carefully determined.

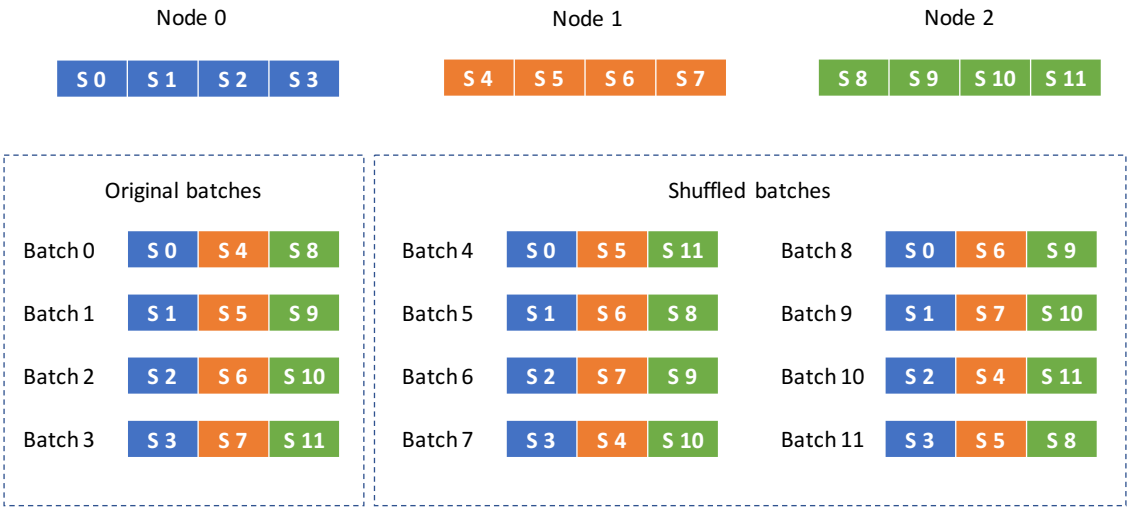

Figure 8.5: Intra-process subbatch shuffling

Intra-process subbatch shuffling is illustrated in Figure 8.5 (S is a subbatch). Suppose there are three nodes in the system, one process per node, and four subbatches in the memory of each process. In the state-of-art data reading, which also includes LMDBIO, only four batches can be created from the in-memory data (i.e., batches inside the "Original batches" box in Figure 8.5). Moreover, the data is typically discarded from memory immediately after the training of each batch is completed. In our approach, we trade smaller reuse distance for lesser data movement. The data is reused while it is still in memory, which is earlier than one epoch. Thus, our reuse distance is smaller than the one in general practice. In this particular example, we can generate at least eight more batches using the same data (i.e., batches inside the "Shuffled batches" box in Figure 8.5).

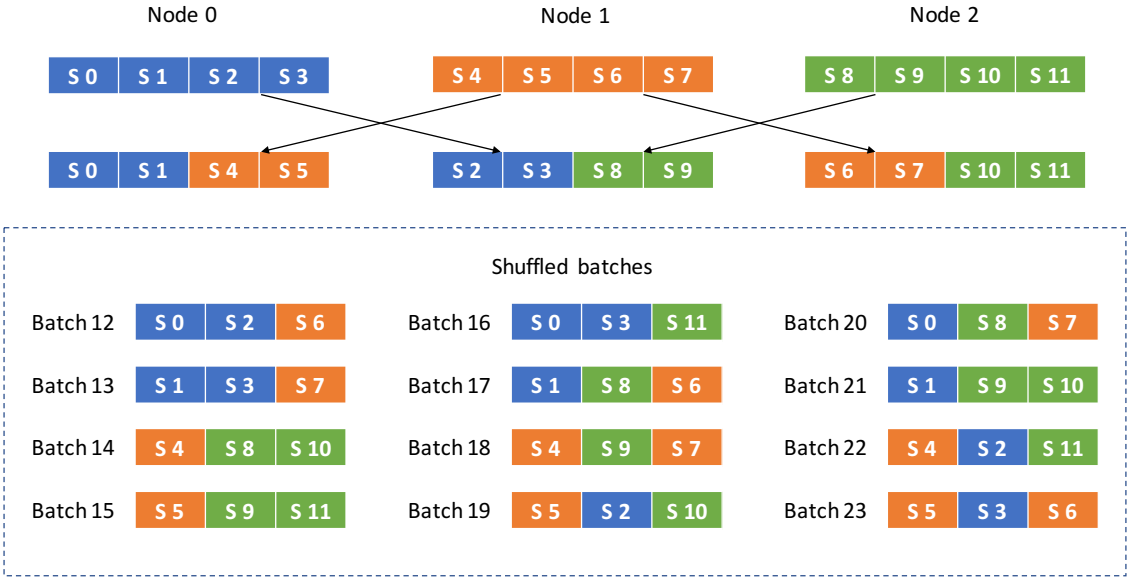

Figure 8.6: Inter-process subbatch shuffling

Inter-process subbatch shuffling, which includes all the levels shown in Table 8.4 except for the first one, is demonstrated in Figure 8.6. There is one additional step to this method from the intra-process subbatch shuffling that is the processes exchange subbatches among one another at the beginning. The remaining steps are as same as the intra-process subbatch shuffling. In this case, we can compose 24 new batches from the data that is already in memory.

Our approach may suffer from the lower convergence rate as the reuse distance can be smaller than the one of the state-of-art approach (i.e., one epoch). Thus, we may need to perform more training iterations to reach the state-of-art level of inference accuracy. However, we expect that the reduced data movement will allow our approach to complete each training iteration faster, which should result in the overall training performance improvement.

Related Work

Most of the efforts in data reuse optimizations in DL focus on minimizing data movement in the processor-memory subsystem to reduce energy consumption by improving the data

flow and data access patterns during the training. Eyeriss [22] maximizes data reuse in the processing element's scratch pads to minimize data access to an on-chip global buffer and an off-chip DRAM of a reconfigurable accelerator. Similarly, some researchers have proposed approaches to maximize data reuse in GPU's register file to reduce the amount of data movement [42, 124]. In contrast, our work focuses on minimizing data movement in the file I/O subsystem which has not been accounted for in these works.

Various researchers have also explored efficient data shuffling methods. DeepIO [191] utilizes multiple process memories, similar to what we have proposed, to store the dataset to enable in-memory data shuffling. However, in the case that a dataset cannot fit in memory, DeepIO fails to minimize the amount of data movement as it only pipelines data reading with the training. For the file I/O bound problems where the computation is light, DeepIO is unlikely to be able to improve the data reading performance of such training. The Distributed In-Memory Data [86] approach also shuffles data in multi-node memory. However, it makes an impractical assumption that the total available memory on all nodes is always sufficient to accommodate the entire dataset. Lightweight Implementation of Random Shuffling (LIRS) [77] leverages a fast storage (i.e., Intel Optane SSD) instead of a main memory to allow for efficient random data accesses. LIRS shuffles indexes of the entire training dataset in memory and directly accesses data samples in the storage based on the randomized indexes. This work is not practical for large-scale systems where the storage system is shared and remote. On a final note, none of the approaches consider reusing input data with the reuse distance smaller than one epoch.

8.4.2 Topology-Aware Parameter Servers for Asynchronous Training

SGD is one of the most common techniques used in the state-of-art DL frameworks. The default SGD approach updates trainable parameters at the end of each training iteration to continually improve the quality of the model. This type of training is referred to as "synchronous" training. Although synchronous SGD training [192] has been proven to be highly efficient as it can provide good inference accuracy as well as good convergence rate, its bulk synchronous nature prevents it from scaling onto large-scale computer systems due to the communication bottleneck.

To alleviate the network I/O bottleneck, an asynchronous version of SGD was introduced. This training model updates parameters asynchronously in every iteration. In other words, the processes are not blocked during the parameter update step, which allows the training of an individual process to progress independently from one another and results in an improvement of overall training progress. Because of the asynchronous parameter update nature, the parameters can be updated only partially which makes the training model susceptible to training divergence as well as low inference accuracy. Asynchronous training is generally implemented using parameter servers [30, 35, 96, 97, 98, 187, 188], where parameters are read and write through centralized servers.

Figure 8.7 shows one possible structure of a connection of workers and parameter servers, the fat-tree topology which is widely adopted in supercomputer systems (W is a worker, and PS is a parameter server). In the case of data parallel training, every worker has to access every parameter server during the parameter update step irrespective of how far the worker is from the parameter server. The cost of accessing the parameter server of each worker is non-uniform since the distances between the pairs of workers and parameter servers are

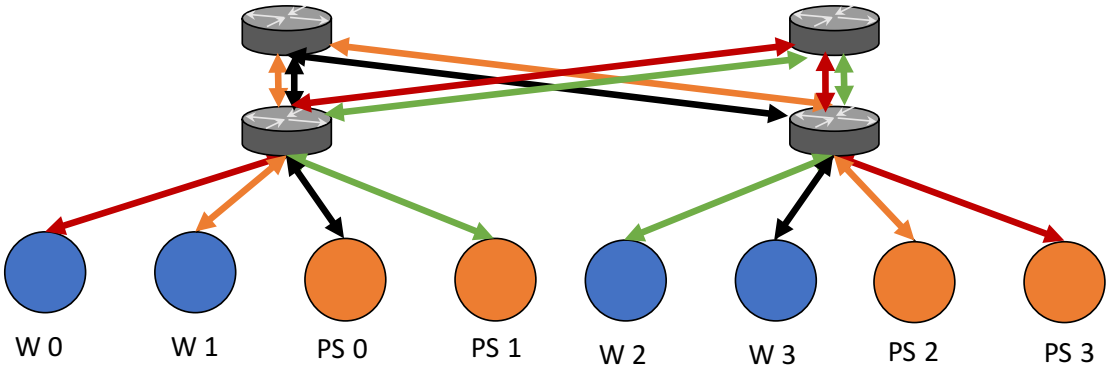

Figure 8.7: Connection to parameter servers (fat-tree topology)

different. As discussed in Chapter 1.2, the cost of moving data, regarding time and energy, increases with the range of which data is moved. For instance, it is more expensive for $W0$ to access $PS3$ than $PS0$ since $W0$ and $P3$ are connected to a different leaf switch. Without network I/O congestion, the parameter update step of an individual worker is already expensive. With a large amount of global communication, the network I/O is highly congested, which makes the parameter update of each worker even more expensive. Note that although the fat-tree topology is one of the interconnections that are good for hiding contention compared to the torus (e.g., in Blue Gene systems—5D torus) and Dragonfly (e.g., in Cray supercomputers—Cray Aries) structures, from Figure 8.7, high network I/O contention can still be observed.

Therefore, we propose to tackle the parameter server access problem by reducing the amount of global data movement. One of the possible methods to solve this problem is to have additional processes as proxies that communicate with a subset of workers and a subset of parameter servers. The purpose of these proxies is to localize the communication to reduce the global data transfer congestion. To access the parameter servers, the worker processes communicate with only their local proxy, which is the nearest one. The hierarchical structure of the proxies is based on the machine and network I/O topologies, which can be realized

through `hwloc` [19] and `netloc` [51].

For each machine, we take into account the internal hierarchical topology of the machine, such as non-uniform memory access (NUMA) memory nodes, sockets, shared caches, and cores, to determine the structure of the proxies. The design principle of the on-node proxy structure is to reduce the data movement cost within the node. For the NUMA architecture, the memory access cost is minimized if the process accesses the nearest memory to the NUMA node of the core that it is running on. Usually, interprocess communication is carried out through shared memory. Thus, processes on the same NUMA node can communicate with one another with a smaller cost than with the ones on the different nodes [99]. Therefore, to optimize the data movement cost, each NUMA node can have one proxy, which is a process that is running on or bound to one of the cores inside that NUMA node, as shown in Figure 8.8.

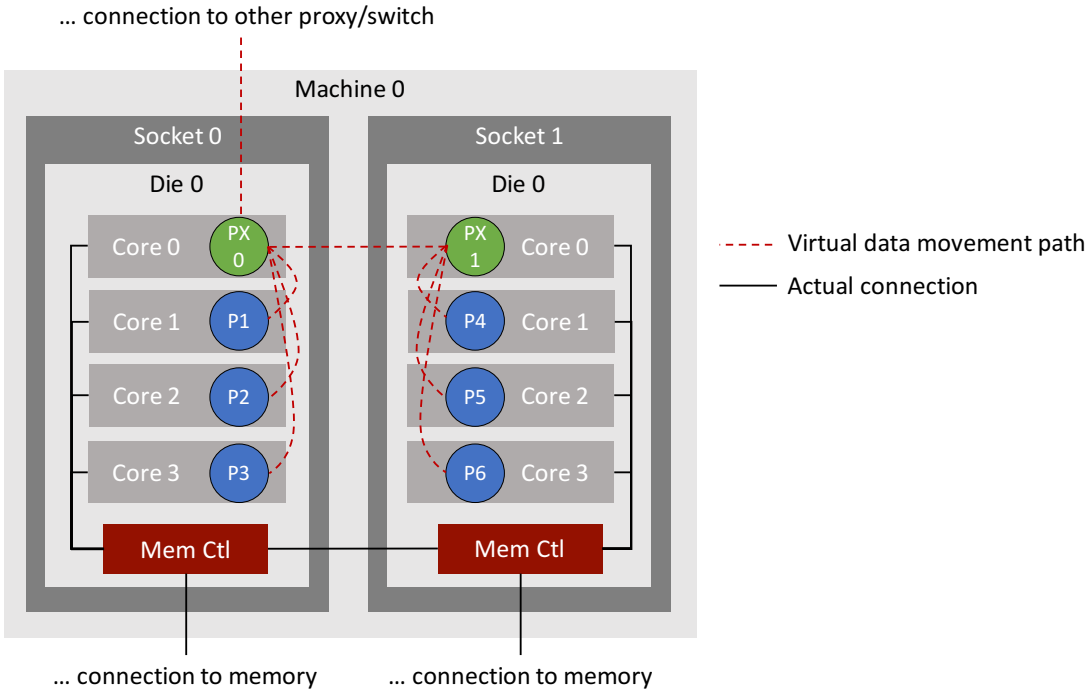

Figure 8.8: On-node non-uniform memory access (NUMA) topology with proxies

Figure 8.8 demonstrates a machine with two NUMA nodes ($MemCtl$ is a memory controller, PX is a proxy process, and P is a computing process). Each NUMA node has four cores, and each core has one process, either a proxy or a computing process, running on. The solid black line represents the physical connection between a core and a memory controller or between the memory controllers, which is where the actual data transfer happens. However, we show the red dotted line to demonstrate the virtual data movement between processes. In this case, all computing processes on the same NUMA node send/receive data to/from the proxy process. Since there are multiple proxies within the machine, one of the proxies acts as a root proxy, which is responsible for communicating with external proxies. We henceforth call the root proxy the "on-node proxy".

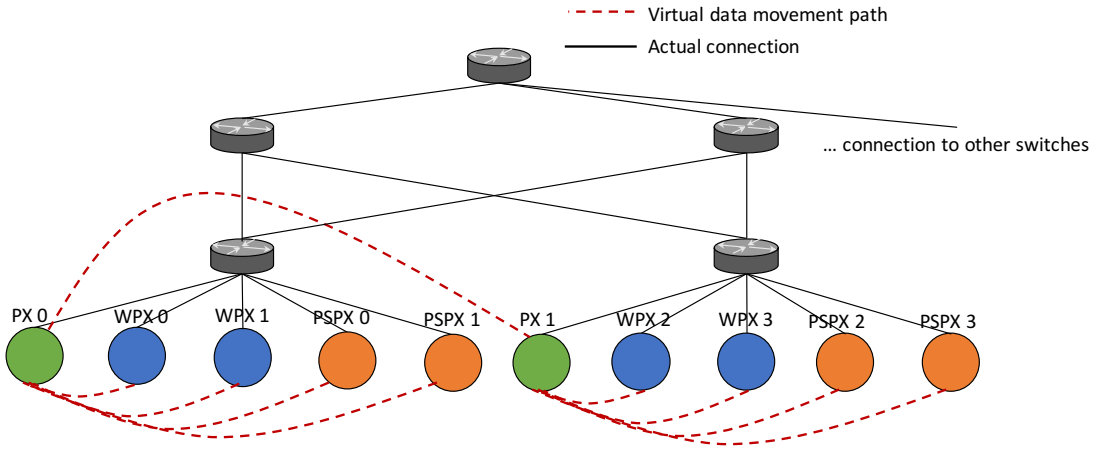

Figure 8.9: Parameter servers with proxies

For the off-node topology, we can localize the data movement by having one proxy per a leaf switch. The off-node proxy can be one of the on-node proxies or a new process on one of the nodes on the same switch. This model is illustrated in Figure 8.9 ($PSPX$ is an on-node proxy of the parameter server, WPX is an on-node proxy of the worker, PX is an off-node proxy, a solid black line is a physical data movement path, and a dotted red line is a virtual data movement path). Although we show the connection of the on-node and off-node proxies in the figure, for conciseness, we further refer to the on-node proxy of the parameter server,

the on-node proxy of the worker, and the off-node proxy as parameter server, worker, and proxy, respectively. To write to the parameter server, the proxy locally reduces parameters from the workers and forwards the accumulated results to the parameter server directly or to another proxy. To read from the parameter server, the proxy fetches data from the server or another proxy and forward it to the corresponding workers. Note that the proxy can prefetch parameters before they receive actual requests from the workers.

Although our proposed approach increases the total amount of communication, most of the data movement is local. In other words, we trade more local data movement for lesser global communication, which, based on our hypothesis, can improve communication as a whole. With our solution, the current parameter update policy has to be modified to enable the proxies to prefetch parameters from the servers or hold parameters for reduction before for updating them on the parameter servers. We may have to perform the training with higher iteration counts than the typical approach as our new parameter update scheme can result in a lower convergence rate. However, we expect to improve the parameter update performance significantly. Therefore, our proposed work should deliver better overall training performance.

Related Work

A large body of work has been proposed to increase training capability and scalability of DL using parameter servers. The work in this area focuses on maximizing data consistency and minimizing overall data movement. However, none of the work has taken into account the topology of the parameter servers in their optimizations.

Google [38, 98] has implemented large-scale asynchronous training using the parameter server model and proposed a training algorithm, namely DownpourSGD, that works on top of the model. DownpourSGD adopts Adagrad [41], a method to use a separate adaptive learning

rate for each parameter, to address the inconsistent parameter update problem. It has been reported to provide good inference accuracy and training convergence rate for the speech recognition application (with 42M parameters) using nearly 2K CPU cores. Petuum [174], a general purpose parameter server, adopts the bounded-asynchronous consistency model to preserve data-parallel convergence guarantees. It can handle up to 220B parameters of the YahooLDA [8] application using 256 cores. Apache PS-Lite [96] is a key-value store parameter server model that is used in MXNet. It can handle petabytes of data with billions of data samples and parameters. The critical optimizations in PS-Lite, are mainly related to data movement optimizations, include (1) range push and pull: for using a small number of large messages instead of a large number of small messages, (2) user-defined filter: for minimizing amount of computation (e.g., transferring only non-zero values), and (3) message compression: for reducing message size. Although the experiments were run on large scale systems (up to 1,000 nodes; 16 cores on each node), the scalability of PS-Lite has not been reported in the literature.

8.4.3 Dynamic Batch Sizing

We have presented in Chapter 7.4 that a large batch is used in common practice; however, it can degrade model quality. To support this claim, we train the original CIFAR10 dataset (50,000 images in an epoch) on AlexNet by using different batch sizes. The inference accuracy of each batch size is shown in Figure 8.10.

From the experimental results, the batch size of 64 is the optimal one as it is the smallest batch size that provides the highest inference accuracy. Although the larger batch sizes achieve slightly worse convergence accuracies, they allow for more computation parallelism. We can leverage such an advantage by adding more resources to accelerate the training

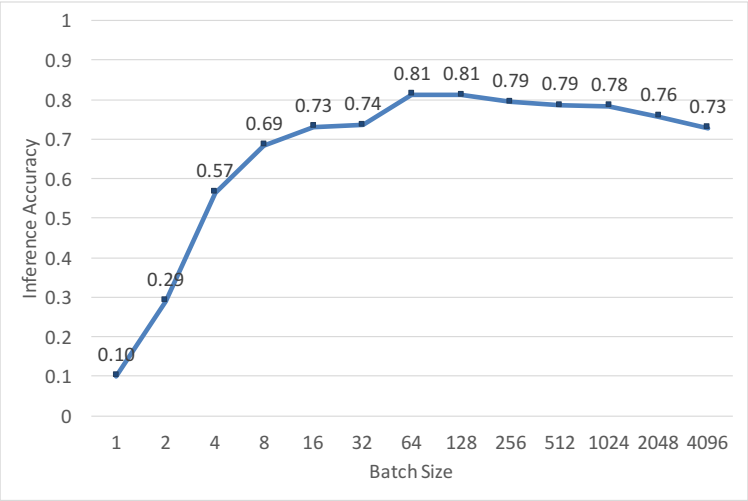

Figure 8.10: Batch size vs. inference accuracy (CIFAR10-AlexNet training)

convergence. In fact, even for typical sequential training, large batch sizes provide better training throughput than the small batch sizes. The improvement of throughput for large batch sizes is subject to less parameter update frequency.

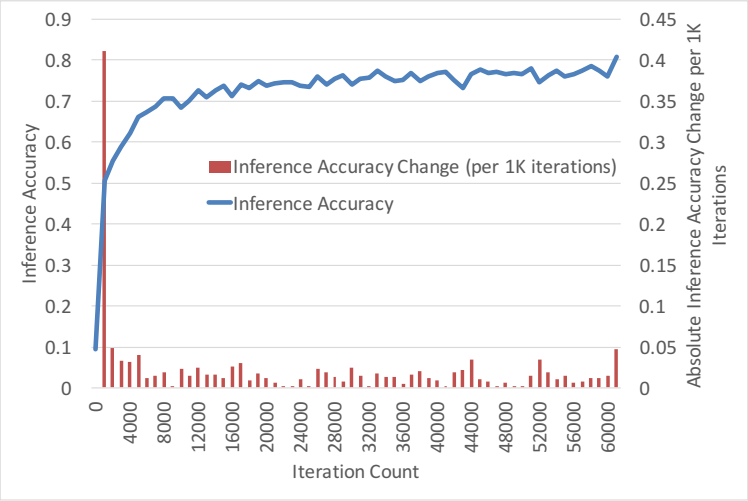

Figure 8.11: Inference accuracy vs. iteration count for the batch size of 64

From the same experiment, we plot the inference accuracy of the batch size of 64 against an iteration count as shown in Figure 8.11. We also compute the absolute inference accuracy change per one thousand iterations and plotted it in the same graph. Based on the graph,

the inference accuracy changes dramatically in the first few thousands of iterations. After the 5,000th iteration (i.e., approximately 6.4 epochs; 3.2 million images), the inference accuracy begins to converge and gradually increases with the average change of 0.01 for every 1,000 iterations, which is only 0.00001 per iteration. The average change of inference accuracy between iterations is extremely small making the high-frequency updates unnecessary. Therefore, the training can reduce parameter update frequency after the training starts to converge.

Adjusting parameter update frequency is practical as it is based on the principle that the inference accuracy slowly becomes more robust to low-quality parameter changes as the training progresses. The parameter changes at the beginning of the training are highly sensitive as the rest of parameter updates are based on them [170]. To ensure that the training moves into the direction of the true gradient (i.e., towards the minimum error), we need to be careful not to make dramatic moves at the start (i.e., using small learning rate and/or batch size) so that we can correct the parameter change direction in the next iteration in the case that the current move is not appropriate. Once the training moves towards the true gradient direction, we can aggressively change the parameters (i.e., using large learning rate and/or batch size).

In this proposed work, we attempt to balance inference accuracy and training speed by using the dynamic batch sizing approach. We start the training by using a small or medium size batch. Then, we periodically evaluate the progress of the training and increase the size of the batch along with the amount of underlying compute resources when appropriate. Doing so increases computation capability which enables the training to process data samples in a shorter amount of time; thus, allowing the training to reach the convergence and the target inference accuracy faster. Moreover, increasing the size of the batch also reduces the amount of process synchronization since we train a larger number of data samples before updating

parameters. Therefore, the overall data movement during parameter updates is minimized; hence, increasing data processing throughput.

Our approach scales the batch size based on two aspects of the training progress.

1. **Convergence rate**: We scale the batch size up once the convergence rate is smaller than a preset threshold.

2. **Percentage of reused data in the batch (PRD)**: In the case that the data reuse optimization (presented in Chapter 8.4.1) is used in collaboration with the dynamic batch sizing approach, we can consider PRD in our batch size scaling policy. Suppose N is a number of data samples that we read to memory each time, B is an initial batch size, where $B < N$, and N is a multiple of B ($N \bmod B = 0$). We can gradually increase the batch size, starting from B, when the PRD reaches some threshold. Once the batch size reaches N, we read new batches of data to memory and scale batch size down to B. Note that this scaling policy is a saw-tooth-like scaling rather than a monotonic increase.

Large batch sizes have a negative impact on the inference accuracy but have a positive impact on the data processing speed due to increased parallelism. Since the rate of convergence of the inference accuracy reduces as the training progresses, we hypothesize that using smaller batch sizes early during the training and large batch sizes later in the training would allow us to benefit from the faster data processing of large batch sizes without being significantly affected by the negative influence on the inference accuracy.

Related Work

Google Brain [147] has proposed the static batch size scaling scheme that is to increase the batch size after every certain number of iterations. Although the scaling policy is simple, it

is not practical as a convergence of parameters is rarely linear in practice. Moreover, such an approach assumes prior knowledge on training convergence. For instance, from one of the experiments, the batch size of the ImageNet training is doubled at the 30^{th} epoch as they know beforehand that the training begins to converge at that epoch. Although this batch size scaling policy works well with the training of ImageNet, it might not deliver the same performance for the training of other datasets and neural networks. Ignoring the progress of the training can deviate the direction of the parameter update into the wrong way, which can result in training divergence.

Bibliography

[1] GRPC: A High Performance, Open-Source Universal RPC Framework. http://www.grpc.io.

[2] TensorFlow Architecture. https://www.tensorflow.org/extend/architecture.

[3] Caffe-MPI for Deep Learning. https://github.com/Caffe-MPI/Caffe-MPI.github.io, September 2015.

[4] MPI: A Message-Passing Interface Standard. http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, September 2015.

[5] Cerebras Reveals World's 'Largest Computer Chip' for AI Tasks. https://www.bbc.com/news/technology-49395577, August 2019.

[6] 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, Dan 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. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[7] Martín Abadi, Michael Isard, and Derek G Murray. A Computational Model for

TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–7. ACM, 2017.

[8] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J Smola. Scalable Inference in Latent Variable Models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining*, pages 123–132. ACM, 2012.

[9] Alham Fikri Aji and Kenneth Heafield. Sparse Communication for Distributed Gradient Descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 440–445, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[10] Ashwin M Aji, Lokendra S Panwar, Feng Ji, Milind Chabbi, Karthik Murthy, Pavan Balaji, Keith R Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, et al. On the Efficacy of GPU-integrated MPI for Scientific Applications. In *Proceedings of the 22nd International Symposium on High-performance Parallel and Distributed Computing*, pages 191–202. ACM, 2013.

[11] Ashwin M Aji, Lokendra S Panwar, Feng Ji, Karthik Murthy, Milind Chabbi, Pavan Balaji, Keith R Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, et al. MPI-ACC: Accelerator-aware MPI for Scientific Applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1401–1414, 2015.

[12] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely Large Minibatch SGD: Training Resnet-50 on ImageNet in 15 Minutes. *arXiv preprint arXiv:1711.04325*, 2017.

[13] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. QSGD:

Communication-Efficient SGD via Gradient Quantization and Encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720, 2017.

[14] A. A. Awan, J. Bdorf, C. Chu, H. Subramoni, and D. K. Panda. Scalable Distributed DNN Training using TensorFlow and CUDA-Aware MPI: Characterization, Designs, and Performance Evaluation. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 498–507, May 2019.

[15] Ammar Ahmad Awan, Khaled Hamidouche, Jahanzeb Maqbool Hashmi, and Dhabaleswar K Panda. 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*, pages 193–205. ACM, 2017.

[16] Yoshua Bengio. Practical Recommendations for Gradient-based Training of Deep Architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer, 2012.

[17] André B Bondi. Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2nd International Workshop on Software and Performance*, pages 195–203. ACM, 2000.

[18] Antoine Bordes, Sumit Chopra, and Jason Weston. Question Answering with Subgraph Embeddings. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 615–620, Doha, Qatar, October 2014. Association for Computational Linguistics.

[19] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. Hwloc: A Generic Framework For Managing Hardware Affinities In HPC Applications. In

Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pages 180–186. IEEE, 2010.

[20] Chia-Yu Chen, Jungwook Choi, Daniel Brand, Ankur Agrawal, Wei Zhang, and Kailash Gopalakrishnan. AdaComp: Adaptive Residual Gradient Compression for Data-Parallel Distributed Training. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

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

[22] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An Energy-efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

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

[24] Wei Der Chien. An Evaluation of TensorFlow as a Programming Framework for HPC Applications. Master's thesis, KTH Royal Institute of Technology, Sweden, 2018.

[25] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, volume 14, pages 571–582, 2014.

[26] Minsik Cho, Ulrich Finkler, Sameer Kumar, David Kung, Vaibhav Saxena, and Dheeraj Sreedhar. PowerAI DDL. *arXiv preprint arXiv:1708.02188*, 2017.

[27] Howard Chu. Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/.

[28] Dan CireşAn, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. Multi-column Deep Neural Network for Traffic Sign Classification. *Neural Networks*, 32:333–338, 2012.

[29] Alexander Collins, Christian Fensch, Hugh Leather, and Murray Cole. MaSiF: Machine Learning Guided Auto-tuning of Parallel Skeletons. In *20th Annual International Conference on High Performance Computing*, pages 186–195. IEEE, 2013.

[30] Henggang Cui, Hao Zhang, Gregory R Ganger, Phillip B Gibbons, and Eric P Xing. GeePS: Scalable Deep Learning on Distributed GPUs with a GPU-specialized Parameter Server. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 4. ACM, 2016.

[31] Xuewen Cui and Wu-chun Feng. Iterative Machine Learning (IterML) for Effective Parameter Pruning and Tuning in Accelerators. In *16th ACM International Conference on Computing Frontiers*, Alghero, Sardinia, Italy, April 2019.

[32] Xuewen Cui, Thomas RW Scogland, Bronis R de Supinski, and Wu-chun Feng. Directive-based Pipelining Extension for OpenMP. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 481–484. IEEE, 2016.

[33] Xuewen Cui, Thomas RW Scogland, Bronis R de Supinski, and Wu-chun Feng. Directive-based Partitioning and Pipelining for Graphics Processing Units. In *2017*

IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 575–584. IEEE, 2017.

[34] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-standard API for Shared-memory Programming. *Computing in Science & Engineering*, (1):46–55, 1998.

[35] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth A Gibson, and Eric P Xing. High-Performance Distributed ML at Scale through Parameter Server Consistency Models. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 79–87, 2015.

[36] Anwesha Das, Frank Mueller, Charles Siegel, and Abhinav Vishnu. Desh: Deep Learning for System Health Prediction of Lead Times to Failure in HPC. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 40–51. ACM, 2018.

[37] Glenn Davis and Russ Rew. Data Management: NetCDF: An Interface for Scientific Data Access. *IEEE Computer Graphics and Applications*, 10:76–82, 1990.

[38] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large Scale Distributed Deep Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[39] N. Dryden, N. Maruyama, T. Moon, T. Benson, A. Yoo, M. Snir, and B. Van Essen. Aluminum: An Asynchronous, GPU-Aware Communication Library Optimized for Large-Scale Training of Deep Neural Networks on HPC Systems. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 1–13, November 2018.

[40] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication Quantization for Data-parallel Training of Deep Neural Networks. In *Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[41] John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[42] Sindhuja Gopalakrishnan Elango. *Convolutional Neural Network Acceleration on GPU by Exploiting Data Reuse*. PhD thesis, San Jose State University, 2017.

[43] Facebook. Gloo. https://github.com/facebookincubator/gloo/blob/master/docs/readme.md, 2017.

[44] Thomas L Falch and Anne C Elster. Machine Learning Based Auto-tuning for Enhanced OpenCL Performance Portability. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 1231–1240. IEEE, 2015.

[45] Clément Farabet, Camille Couprie, Laurent Najman, and Yann LeCun. Scene Parsing with Multiscale Feature Learning, Purity Trees, and Optimal Covers. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, ICML12, page 18571864, Madison, WI, USA, 2012. Omnipress.

[46] Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. Large-scale FPGA-based Convolutional Networks. *Scaling Up Machine Learning: Parallel and Distributed Approaches*, pages 399–419, 2011.

[47] Michael Feldman. Intel Spills Details on Knights Mill Processor. https://www.top500.org/news/intel-spills-details-on-knights-mill-processor/, 2017.

[48] Christophe Garcia and Manolis Delakis. Convolutional Face Finder: A Neural Architecture for Fast and Robust Face Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(11):1408–1423, 2004.

[49] Joseph D Garvey and Tarek S Abdelrahman. Automatic Performance Tuning of Stencil Computations on GPUs. In *2015 44th International Conference on Parallel Processing*, pages 300–309. IEEE, 2015.

[50] Andrew Gibiansky. Bringing HPC Techniques to Deep Learning. http://andrew.gibiansky.com.

[51] Brice Goglin, Joshua Hursey, and Jeffrey M Squyres. Netloc: Towards a Comprehensive View of the HPC System Topology. In *2014 43rd International Conference on Parallel Processing Workshops (ICCPW)*, pages 216–225. IEEE, 2014.

[52] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[53] Google. Cloud Tensor Processing Units (TPUs). https://cloud.google.com/tpu/docs/tpus, 2018.

[54] Google. gRPC over HTTP2. https://github.com/grpc/grpc/blob/master/doc/PROTOCOL-HTTP2.md, 2018.

[55] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. *arXiv preprint arXiv:1706.02677*, 2017.

[56] Richard L Graham, Devendar Bureddy, Pak Lui, Hal Rosenstock, Gilad Shainer, Gil Bloch, Dror Goldenerg, Mike Dubman, Sasha Kotchubievsky, Vladimir Koushnir, et al. Scalable Hierarchical Aggregation Protocol (SHArP): A Hardware Architecture for

Efficient Data Reduction. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 1–10. IEEE, 2016.

[57] Ivan Grasso, Klaus Kofler, Biagio Cosenza, and Thomas Fahringer. Automatic Problem Size Sensitive Task Partitioning on Heterogeneous Parallel Systems. In *ACM SIGPLAN Notices*, volume 48, pages 281–282. ACM, 2013.

[58] The HDF Group. Enabling a Strict Consistency Semantics Model in Parallel HDF5. https://support.hdfgroup.org/HDF5/doc/Advanced/PHDF5FileConsistencySemantics/PHDF5FileConsistencySemantics.pdf, 2012.

[59] Jeff Hale. Which Deep Learning Framework is Growing Fastest? https://towardsdatascience.com/which-deep-learning-framework-is-growing-fastest-3f77f14aa318, 2019.

[60] Jiawei Han, Jian Pei, and Micheline Kamber. *Data Mining: Concepts and Techniques*. Elsevier, 2011.

[61] Mark Harris. NVIDIA DGX-1: The Fastest Deep Learning System. https://devblogs.nvidia.com/dgx-1-fastest-deep-learning-system/, 2017.

[62] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, December 2015.

[63] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[64] Moritz Helmstaedter, Kevin L Briggman, Srinivas C Turaga, Viren Jain, H Sebastian

Seung, and Winfried Denk. Connectomic Reconstruction of the Inner Plexiform Layer in the Mouse Retina. *Nature*, 500(7461):168, 2013.

[65] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised First Edition*. Morgan Kaufmann, 2012.

[66] J. Hines. Stepping up to Summit. *Computing in Science Engineering*, 20(2):78–82, Mar 2018.

[67] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. Process-in-Process: Techniques for Practical Address-Space Sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, pages 131–143. ACM, 2018.

[68] Forrest N Iandola, Matthew W Moskewicz, Khalid Ashraf, and Kurt Keutzer. FireCaffe: Near-linear Acceleration of Deep Neural Network Training on Compute Clusters. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2592–2600, 2016.

[69] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML15, page 448456. JMLR.org, 2015.

[70] Shintaro Iwasaki, Abdelhalim Amer, Kenjiro Taura, Sangmin Seo, and Pavan Balaji. BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 29–42. IEEE, 2019.

[71] Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On Using Very Large Target Vocabulary for Neural Machine Translation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1–10, Beijing, China, July 2015. Association for Computational Linguistics.

[72] John Jenkins, James Dinan, Pavan Balaji, Nagiza F Samatova, and Rajeev Thakur. Enabling Fast, Noncontiguous GPU Data Movement in Hybrid MPI+ GPU Environments. In *2012 IEEE International Conference on Cluster Computing*, pages 468–476. IEEE, 2012.

[73] Feng Ji, Ashwin M Aji, James Dinan, Darius Buntinas, Pavan Balaji, Wu-chun Feng, and Xiaosong Ma. Efficient Intranode Communication in GPU-accelerated Systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 1838–1847. IEEE, 2012.

[74] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly Scalable Deep Learning Training System with Mixed-precision: Training ImageNet in Four Minutes. In *2018 Workshop on Systems for ML and Open Source Software, Thirty-third Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[75] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia*, MM 14, page 675678, New York, NY, USA, 2014. ACM.

[76] Kaggle. Data Science Bowl 2017: Can You Improve Lung Cancer Detection? https://www.kaggle.com/c/data-science-bowl-2017, 2017.

[77] Zhi-Lin Ke, Hsiang-Yun Cheng, and Chai-Lin Yang. LIRS: Enabling Efficient Machine Learning on NVM-based Storage via a Lightweight Implementation of Random Shuffling. *arXiv preprint arXiv:1810.04509*, 2018.

[78] Gokcen Kestor, Roberto Gioiosa, Darren J Kerbyson, and Adolfy Hoisie. Quantifying the Energy Cost of Data Movement in Scientific Applications. In *IEEE International Symposium on Workload Characterization (IISWC), 2013*, pages 56–65. IEEE, 2013.

[79] Nikhil Ketkar. *Introduction to PyTorch*, pages 195–208. Apress, Berkeley, CA, 2017.

[80] Akhmedov Khumoyun, Yun Cui, and Lee Hanku. Spark Based Distributed Deep Learning Framework for Big Data Applications. In *International Conference on Information Science and Communications Technologies (ICISCT)*, pages 1–5. IEEE, 2016.

[81] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *2015 3rd International Conference on Learning Representations (ICLR)*, 2015.

[82] Alex Krizhevsky. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv preprint arXiv:1404.5997*, 2014.

[83] Alex Krizhevsky and Geoffrey Hinton. Learning Multiple Layers of Features from Tiny Images. Technical report, University of Toronto, 2009.

[84] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[85] Sameer Kumar, Victor Bitorff, Dehao Chen, Chiachen Chou, Blake Hechtman, HyoukJoong Lee, Naveen Kumar, Peter Mattson, Shibo Wang, Tao Wang, et al. Scale MLPerf-0.6 models on Google TPU-v3 Pods. *arXiv preprint arXiv:1909.09756*, 2019.

[86] Sameer Kumar, Dheeraj Sreedhar, Vaibhav Saxena, Yogish Sabharwal, and Ashish Verma. Efficient Training of Convolutional Neural Nets on Large Distributed Systems. *arXiv preprint arXiv:1711.00705*, 2017.

[87] Thorsten Kurth, Mikhail Smorkalov, Peter Mendygral, Srinivas Sridharan, and Amrita Mathuriya. TensorFlow at Scale: Performance and productivity analysis of distributed training with Horovod, MLSL, and Cray PE ML. *Concurrency and Computation: Practice and Experience*, 31(16):e4989, 2019.

[88] Thorsten Kurth, Sean Treichler, Joshua Romero, Mayur Mudigonda, Nathan Luehr, Everett Phillips, Ankur Mahesh, Michael Matheson, Jack Deslippe, Massimiliano Fatica, Prabhat, and Michael Houston. Exascale Deep Learning for Climate Analytics. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18, pages 51:1–51:12, Piscataway, NJ, USA, 2018. IEEE Press.

[89] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. Deep Learning at 15PF: Supervised and Semi-Supervised Classification for Scientific Data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM, 2017.

[90] Quoc V Le. Building High-level Features using Large Scale Unsupervised Learning. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013*, pages 8595–8598. IEEE, 2013.

[91] LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey. Deep Learning. *Nature*, 521(7553):436, 2015.

[92] Stefan Lee, Senthil Purushwalkam, Michael Cogswell, David J. Crandall, and Dhruv Batra. Why M Heads Are Better than One: Training a Diverse Ensemble of Deep Networks. *arXiv preprint arXiv:1511.06314*, 2015.

[93] Michael KK Leung, Hui Yuan Xiong, Leo J Lee, and Brendan J Frey. Deep Learning of the Tissue-Regulated Splicing Code. *Bioinformatics*, 30(12):i121–i129, 2014.

[94] Jiangtian Li, Xiaosong Ma, Karan Singh, Martin Schulz, Bronis R de Supinski, and Sally A McKee. Machine Learning Based Online Performance Prediction for Runtime Parallelization and Task Scheduling. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 89–100. IEEE, 2009.

[95] Jianwei Li, Wei keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[96] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*, pages 583–598, 2014.

[97] Mu Li, David G Andersen, Alexander J Smola, and Kai Yu. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[98] Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G Andersen, and Alexander Smola. Parameter Server for Distributed Machine Learning. In *Big Learning NIPS Workshop*, volume 6, page 2. 2013.

[99] Shigang Li, Torsten Hoefler, and Marc Snir. NUMA-Aware Shared-Memory Collective Communication for MPI. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 85–96. ACM, 2013.

[100] Wenqiang Li, Guanghao Jin, Xuewen Cui, and Simon See. An Evaluation of Unified Memory Technology on NVIDIA GPUs. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1092–1098. IEEE, 2015.

[101] Min Lin, Qiang Chen, and Shuicheng Yan. Network in Network. *arXiv preprint arXiv:1312.4400*, 2013.

[102] Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. Deep Gradient Compression: Reducing the Communication Bandwidth for Distributed Training. In *2018 6th International Conference on Learning Representations (ICLR)*, April 2018.

[103] Robert Love. *Linux System Programming: Talking Directly to the Kernel and C Library*. O'Reilly Media, Inc., 2013.

[104] Piotr Luszczek. Hardware for Numerical Libraries. http://www.icl.utk.edu/~luszczek/teaching/courses/2017/mhpc/numlinalghw.pdf, 2017.

[105] He Ma, Fei Mao, and Graham W. Taylor. Theano-MPI: A Theano-Based Distributed Training Framework. *CoRR*, abs/1605.08325, 2016.

[106] P. MacArthur, Q. Liu, R. D. Russell, F. Mizero, M. Veeraraghavan, and J. M. Dennis. An Integrated Tutorial on InfiniBand, Verbs, and MPI. *IEEE Communications Surveys Tutorials*, 19(4):2894–2926, 2017.

[107] Alberto Magni, Christophe Dubach, and Michael O'Boyle. Automatic Optimization of Thread-coarsening for Graphics Processors. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, pages 455–466. ACM, 2014.

[108] KV Manian, AA Ammar, Amit Ruhela, C-H Chu, Hari Subramoni, and Dhabaleswar K Panda. Characterizing CUDA Unified Memory (UM)-Aware MPI Designs on Modern GPU Architectures. In *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, pages 43–52. ACM, 2019.

[109] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Kärnä, Diana Moise, Simon J Pennycook, et al. CosmoFlow: Using Deep Learning to Learn the Universe at Scale. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 819–829. IEEE, 2018.

[110] Pierre Matri, María S Pérez, Alexandru Costan, and Gabriel Antoniu. TỳrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018.

[111] Peter Mattson, Christine Cheng, Cody Coleman, Greg Diamos, Paulius Micikevicius, David Patterson, Hanlin Tang, Gu-Yeon Wei, Peter Bailis, Victor Bittorf, et al. MLPerf Training Benchmark. *arXiv preprint arXiv:1910.01500*, 2019.

[112] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed Precision Training. In *2018 6th International Conference on Learning Representations (ICLR)*, April 2018.

[113] H Mikami, H Suganuma, P U-chupala, Y Tanaka, and Y Kageyama. Massively Distributed SGD: ImageNet/ResNet-50 Training in a Flash. *arXiv preprint arXiv:1811.05233*, 2018.

[114] Ingo Molnar. [Announce] [patch] Modular Scheduler Core and Completely Fair Scheduler [CFS]. https://lwn.net/Articles/230501/.

[115] Douglas C Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2008.

[116] Timothy Prickett Morgan. Machine Learning Gets an InfiniBand Boost with Caffe2. https://www.nextplatform.com/2017/04/19/machine-learning-gets-infiniband-boost-caffe2/, 2017.

[117] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I Jordan. SparkNet: Training Deep Networks in Spark. In *2016 4th International Conference on Learning Representations (ICLR)*, May 2016.

[118] Karl Ni, Roger Pearce, Kofi Boakye, Brian Van Essen, Damian Borth, Barry Chen, and Eric Wang. Large-scale Deep Learning on the YFCC100M Dataset. *arXiv preprint arXiv:1502.03409*, 2015.

[119] NVIDIA. NVIDIA Depp Learning Platform: Giant Leaps in Performance and Efficiency for AI Services, From the Data Center to the Networks Edge. https://images.nvidia.com/content/pdf/inference-technical-overview.pdf, 2018.

[120] William F Ogilvie, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Active Learning Accelerated Automatic Heuristic Construction for Parallel Program Mapping. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 481–482. IEEE, 2014.

[121] Travis E Oliphant. *A Guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[122] Kazuki Osawa, Yohei Tsuji, Yuichiro Ueno, Akira Naruse, Rio Yokota, and Satoshi Matsuoka. Large-Scale Distributed Second-Order Optimization Using Kronecker-Factored Approximate Curvature for Deep Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 12359–12367, 2019.

[123] Dhinakaran Pandiyan. *Data Movement Energy Characterization of Emerging Smartphone Workloads for Mobile Platforms*. Arizona State University, 2014.

[124] Hyunsun Park, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. Zero and Data Reuse-Aware Fast Convolution for Deep Neural Networks on GPU. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 33. ACM, 2016.

[125] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An Imperative Style, High-performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[126] Pitch Patarasuk and Xin Yuan. Bandwidth Optimal All-reduce Algorithms for Clusters of Workstations. *Journal of Parallel and Distributed Computing*, 69(2):117–124, 2009.

[127] Simone Pellegrini, Thomas Fahringer, Herbert Jordan, and Hans Moritsch. Automatic Tuning of MPI Runtime Parameter Settings by Using Machine Learning. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pages 115–116. ACM, 2010.

[128] Sarunya Pumma, Daniele Buono, Fabio Checconi, Xinyu Que, and Wu-chun Feng.

Alleviating Load Imbalance in Data Processing for Large-Scale Deep Learning. In *2020 IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, under review.

[129] Sarunya Pumma, Wu-chun Feng, Phond Phunchongharn, Sylvain Chapeland, and Tiranee Achalakul. A Runtime Estimation Framework for ALICE. *Future Generation Computer Systems*, 72:65–77, 2017.

[130] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Parallel I/O Optimizations for Scalable Deep Learning. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 720–729. IEEE, 2017.

[131] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. Towards Scalable Deep Learning via I/O Analysis and Optimization. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 223–230. IEEE, 2017.

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

[133] Carl Edward Rasmussen. Gaussian Processes in Machine Learning. In *Summer School on Machine Learning*, pages 63–71. Springer, 2003.

[134] Baidu Research. baidu-allreduce. https://github.com/baidu-research/baidu-allreduce.

[135] Microsoft Research. The Microsoft Cognitive Toolkit. https://docs.microsoft.com/en-us/cognitive-toolkit/, 2017.

[136] Stephen J. Rogowski. Bus. In *Encyclopedia of Computer Science*, pages 165–167. John Wiley and Sons Ltd., Chichester, UK.

[137] Karl Rupp. Microprocessor Trend Data. https://github.com/karlrupp/microprocessor-trend-data, 2018.

[138] 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. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[139] Tara N Sainath, Abdel-rahman Mohamed, Brian Kingsbury, and Bhuvana Ramabhadran. Deep Convolutional Neural Networks for LVCSR. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2013*, pages 8614–8618. IEEE, 2013.

[140] Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 1-bit Stochastic Gradient Descent and Its Application to Data-parallel Distributed Training of Speech DNNs. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

[141] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, et al. Argobots: A Lightweight Low-level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2017.

[142] Alexander Sergeev and Mike Del Balso. Horovod: Fast and Easy Distributed Deep Learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[143] John Shalf, Sudip Dosanjh, and John Morrison. Exascale Computing Technology

Challenges. In *International Conference on High Performance Computing for Computational Science*, pages 1–25. Springer, 2010.

[144] Hongzhang Shan and John Shalf. Using IOR to Analyze the I/O Performance for HPC Platforms. https://escholarship.org/uc/item/9111c60j, 2007.

[145] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *2015 3rd International Conference on Learning Representations (ICLR)*, 2015.

[146] Karan Singh, Engin İpek, Sally A McKee, Bronis R de Supinski, Martin Schulz, and Rich Caruana. Predicting Parallel Application Performance via Machine Learning Approaches. *Concurrency and Computation: Practice and Experience*, 19(17):2219–2235, 2007.

[147] Samuel L Smith, Pieter-Jan Kindermans, and Quoc V Le. Don't Decay the Learning Rate, Increase the Batch Size. In *2018 6th International Conference on Learning Representations (ICLR)*, April 2018.

[148] Facebook Open Source. Caffe2 A New Lightweight, Modular, and Scalable Deep Learning Framework. https://caffe2.ai.

[149] Srinivas Sridharan, Karthikeyan Vaidyanathan, Dhiraj Kalamkar, Dipankar Das, Mikhail E Smorkalov, Mikhail Shiryaev, Dheevatsa Mudigere, Naveen Mellempudi, Sasikanth Avancha, Bharat Kaul, et al. On Scale-out Deep Learning Training for Cloud and HPC. In *SysML Conference*, February 2018.

[150] Nikko Strom. Scalable Distributed DNN Training Using Commodity GPU Cloud Computing. In *Sixteenth Annual Conference of the International Speech Communication Association*, 2015.

[151] Jeff A Stuart, Pavan Balaji, and John D Owens. Extending MPI to Accelerators. In *Proceedings of the 1st Workshop on Architectures and Systems for Big Data*, pages 19–23. ACM, 2011.

[152] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[153] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2818–2826, 2016.

[154] Yaniv Taigman, Ming Yang, Marc' Aurelio Ranzato, and Lior Wolf. Deepface: Closing the Gap to Human-level Performance in Face Verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1701–1708, 2014.

[155] TensorFlow. How To Compile, Use and Configure RDMA-Enabled TensorFlow. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/verbs/README.md.

[156] R. Thakur, W. Gropp, and E. Lusk. A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *IEEE/ACM Conference on Supercomputing (SC)*, November 1998.

[157] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, Washington, DC, USA, 1999.

[158] Rajeev Thakur, Ewing Lusk, and William Gropp. Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. Technical report, Technical Report ANL/MCS-TM-234, Mathematics and Computer Science Division, Argonne National Laboratory, 1997.

[159] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, February 2005.

[160] The Ohio State University. MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu, 2014.

[161] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[162] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A Genetic Algorithms Approach to Modeling the Performance of Memory-bound Computations. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 47. ACM, 2007.

[163] Philippe Tillet and David Cox. Input-aware Auto-tuning of Compute-bound HPC Kernels. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 43. ACM, 2017.

[164] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: A Next-Generation Open Source Framework for Deep Learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6, 2015.

[165] Srinivas C Turaga, Joseph F Murray, Viren Jain, Fabian Roth, Moritz Helmstaedter,

Kevin Briggman, Winfried Denk, and H Sebastian Seung. Convolutional Networks Can Learn to Generate Affinity Graphs for Image Segmentation. *Neural Computation*, 22(2):511–538, 2010.

[166] Abhinav Vishnu, Charles Siegel, and Jeffrey Daily. Distributed TensorFlow with MPI. *CoRR*, abs/1603.02339, 2016.

[167] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[168] Hao Wang, Jing Zhang, Da Zhang, Sarunya Pumma, and Wu-chun Feng. PaPar: A Parallel Data Partitioning Framework for Big Data Applications. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 605–614. IEEE, 2017.

[169] Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Terngrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. In *Advances in Neural Information Processing Systems*, pages 1509–1519, 2017.

[170] D Randall Wilson and Tony R Martinez. The General Inefficiency of Batch Training for Gradient Descent Learning. *Neural Networks*, 16(10):1429–1451, 2003.

[171] Michael Woodacre, Derek Robb, Dean Roe, and Karl Feind. The SGI® AltixTM 3000 Global Shared-memory Architecture. *Silicon Graphics, Inc*, 2005.

[172] Kai Wu, Frank Ober, Shari Hamlin, and Dong Li. Early Evaluation of Intel Optane Non-volatile Memory with HPC I/O Workloads. *arXiv preprint arXiv:1708.02199*, 2017.

[173] Xingfu Wu, Valerie Taylor, Justin M Wozniak, Rick Stevens, Thomas Brettin, and Fangfang Xia. Performance, Energy, and Scalability Analysis and Improvement of Parallel Cancer Deep Learning CANDLE Benchmarks. In *Proceedings of the 48th International Conference on Parallel Processing*, page 78. ACM, 2019.

[174] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. Petuum: A New Platform for Distributed Machine Learning on Big Data. *IEEE Transactions on Big Data*, 1(2):49–67, 2015.

[175] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance Analysis of NVMe SSDs and Their Implication on Real World Databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM, 2015.

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

[177] Chih-Chieh Yang and Guojing Cong. Accelerating Data Loading in Deep Neural Network Training. *arXiv preprint arXiv:1910.01196*, 2019.

[178] Joe Yaworski. Intel Omni-Path Architecture Enables Deep Learning Training on HPC. https://itpeernetwork.intel.com/intel-omni-path-deep-learning-training/, 2017.

[179] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. Towards Zero Copy Dataflows using RDMA. In *Proceedings of the SIGCOMM Posters and Demos*, pages 28–30. ACM, 2017.

[180] Chris Ying, Sameer Kumar, Dehao Chen, Tao Wang, and Youlong Cheng. Image Classification at Supercomputer Scale. In *2018 Workshop on Systems for ML and Open Source Software, Thirty-third Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

[181] Yang You, Igor Gitman, and Boris Ginsburg. Scaling SGD Batch Size to 32k for ImageNet Training. *arXiv preprint arXiv:1708.03888*, 2017.

[182] Yang You, Zhao Zhang, James Demmel, Kurt Keutzer, and Cho-Jui Hsieh. ImageNet Training in 24 Minutes. *arXiv preprint arXiv:1709.05011*, 2017.

[183] Yang You, Zhao Zhang, Cho-Jui Hsieh, James Demmel, and Kurt Keutzer. ImageNet Training in Minutes. In *Proceedings of the 47th International Conference on Parallel Processing (ICPP)*, 2018.

[184] Sergey Zagoruyko and Nikos Komodakis. Wide residual Networks. In Edwin R. Hancock Richard C. Wilson and William A. P. Smith, editors, *Proceedings of the British Machine Vision Conference (BMVC)*, pages 87.1–87.12. BMVA Press, September 2016.

[185] Matthew D Zeiler. ADADELTA: An Adaptive Learning Rate Method. *arXiv preprint arXiv:1212.5701*, 2012.

[186] Kunlei Zhang and Xue-Wen Chen. Large-Scale Deep Belief Nets with MapReduce. *IEEE Access*, 2:395–403, 2014.

[187] Sixin Zhang, Anna E Choromanska, and Yann LeCun. Deep Learning with Elastic Averaging SGD. In *Advances in Neural Information Processing Systems*, pages 685–693, 2015.

[188] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-Aware Async-SGD for

Distributed Deep Learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI16, pages 2350–2356. AAAI Press, 2016.

[189] Yikai Zhang, Hui Qu, Chao Chen, and Dimitris Metaxas. Taming the Noisy Gradient: Train Deep Neural Networks with Small Batch Sizes. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 4348–4354. AAAI Press, 2019.

[190] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients. *arXiv preprint arXiv:1606.06160*, 2016.

[191] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-Aware I/O Pipelining for Large-Scale Deep Learning on HPC Systems. In *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2018)*, 2018.

[192] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.