AIKIDO: Toward Straggler Mitigation for Distributed Machine Learning Training in Cloud Data Centers

by

Ayush Sharma

S.B., Massachusetts Institute of Technology (2019)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 18, 2020

Certified by .....
Manya Ghobadi
Assistant Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

AIKIDO: Toward Straggler Mitigation for Distributed Machine Learning Training in Cloud Data Centers

by

Ayush Sharma

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As artificial intelligence becomes a critical component of everyday life, the popularity of using cloud data centers for training deep neural networks is relentlessly growing. This poses a significant challenge for data center operators where the network bandwidth is shared among multiple ML jobs as well as between ML jobs and data center flows. At high loads, the network experiences transient congestion events frequently which in turn delays the parameter updates between ML workers. Consequently, the training convergence suffers as some workers behind congested links *straggle* to update the model parameters in time, hence delaying all workers. We propose AIKIDO as a first step towards mitigating the impact of transient network-induced stragglers on training workloads caused by the dynamic nature of the data center traffic. AIKIDO exploits the inherent robustness of ML training on occasional loss of gradient updates and implements a *Skip-Straggler* communication strategy where the updates from straggling workers are simply skipped. In addition, AIKIDO introduces an *Active-Backup* strategy as an improvement to the Skip method to maintain a high accuracy convergence while using fewer resources than full worker replication. In our experiment, we use Google Cloud Engine environment to train ResNet-50 on ImageNet at various scales and demonstrate that AIKIDO is able to mitigate the effect of stragglers and achieve the time-to-accuracy as if there are no stragglers.

Thesis Supervisor: Manya Ghobadi
Title: Assistant Professor

Acknowledgments

I would like to extend my deepest gratitude to my advisor Manya Ghobadi. Her continued support for this project - both academic and professional - was invaluable. Special thanks to my collaborator James Salamy for his help with brain-storming ideas and running experiments. I would also like to express my deepest gratitude to my parents as well as my partner Gabriella Garcia for their ongoing love and support without which I would not be where I am today. Finally, many thanks to MIT, the brilliant professors here, and the extraordinary peers from whom I have learnt a great deal in life.

This research was sponsored in part by MIT's Google Cloud Engine Program and the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

State-of-the-art Machine Learning models increasingly require massive amounts of resources to train [2, 39, 61]. For instance, GPT-2 [47], a large transformer-based language model with 1.5 billion parameters, requires 1000 NVIDIA Tesla V100 GPUs to train under one hour [43], which implies that it will likely take around 1000 hours to converge on one NVIDIA Tesla V100 GPU. Simply put, training large models on one GPU is too slow. Hence, there is an enormous need for efficient *distributed machine learning training* frameworks [23, 18, 41, 25, 42, 53, 3, 35, 4].

The most common approach for distributed training is called data parallelism in which a given neural network is replicated across N workers with each worker processing a small subset of the training data. At every iteration, the local model updates need to be aggregated and distributed across all the workers to ensure model convergence. This step is called *AllReduce* and is often implemented with a parameter-server [31] or Ring-AllReduce [46]. In the parameter-server architecture, one or more parameter-servers holds the current model and synchronizes it between a set of worker-nodes for each iteration [5]. The problem with this approach is that the network links between the parameter-server and the workers become a bottleneck. As a result, the workers cannot utilize their full compute power, while the bandwidth to the parameter-server becomes a bottleneck, slowing down training. To mitigate this problem, Uber recently open-sourced a first-of-its-kind data parallel machine learning framework, called Horovod, which uses Ring-AllReduce to allow users to

distribute their existing machine learning training with minimal code changes at the TensorFlow/Python abstraction layer [52].

Although Ring-AllReduce frameworks improve the training time compared to parameter-server frameworks, they are more sensitive to the problem of *stragglers*. Stragglers are worker nodes that lag behind their peers and add delay to the training [19, 56, 24]. This is because the ring-based communication scheme creates an inter-dependency across workers, hence, the training will proceed at the speed of the slowest worker(s). Stragglers can happen due to several reasons, such as heterogeneous hardware and congested links. Current training frameworks, mitigate hardware-based stragglers by ensuring the training job is launched across workers with the same compute capabilities. However, network-based stragglers, such as workers behind transiently congested links, are much harder to detect and react to given their unpredictable nature. This thesis proposes a solution to mitigate congestion-induced stragglers.

Today's approaches to mitigate network-induced stragglers includes asynchronous distributed training [35]. However asynchronous training frameworks often do not provide the same performance and convergence guarantees as synchronous training [62]. As a result, synchronous Stochastic Gradient Descent (SGD) remains the algorithm of choice for training large-scale models [61]. Recent work proposed using backup workers to mitigate the impact of stragglers under the parameter-server communication pattern [9]. However, as the ML field moves away from parameter-server aggregation mechanism to ring-AllReduce based approaches, there is a need for new mechanisms to provide network-induced straggler protection. This is the opportunity we notice, and therefore we propose AIKIDO as a first step towards straggler protection in large-scale distributed machine learning under allreduce parameter aggregation.

With AIKIDO, we provide a flexible primary-backup architecture under All-Reduce communication pattern to curb the impact of stragglers on training. Building on top of ring-All-Reduce communication pattern, AIKIDO introduces a *Ring-Reshape Module* as well as a *Straggler Simulator* into the shared context of the N distributed ML workers. The Straggler Simulator allows us to inject arbitrary amounts of straggler

behavior into any number of workers in each sync-SGD iteration. The Ring- Reshape Module seamlessly switches the straggler worker out of the aggregation ring for the duration of straggling behavior that is determined by the simulator. We call this approach a *Skip* Strategy. Additionally, we explore an *Active-Backup* strategy where a set of dedicated backup workers are mapped to cover for the straggler workers for a given iteration. These active backup workers can be tuned to work with fewer resources such as quantized model or smaller batch sizes.

We demonstrate the effect of stragglers on training throughput by creating cross traffic TCP flows while training a neural network on real hardware and measuring the impact of network congestion on the workload. Next, we demonstrate the efficacy of AIKIDO in curbing stragglers using upto 16 NVIDIA Tesla V100 workers training a ResNet-50 [20] model with ImageNet dataset [14]. Our results show that, in the face of simulated stragglers, AIKIDO is able to bypass them and achieves the same time-to-convergence as if the stragglers where not present. More specifically, we simulate a 20% straggler rate as follows by adding 75ms of delay to one of the workers on top of the 180ms iteration time for every alternate iteration. On average, this amounts to $\frac{\frac{1}{2}*75*100}{180} = 20\%$ delay to every iteration. As a result, the training job is delayed by approximately 20% in terms of the iteration times, throughput, and time to reach threshold validation accuracy. We show that AIKIDO is able to mitigate the impact of simulated stragglers on par with the ideal run with no simulated stragglers.

The rest of this thesis is organized as follows. First, we cover existing literature and introduce important prior concepts in Chapter 2. Next, we describe our system in depth in Chapter 3. Then, we present and summarize our experimental results in Chapter 4. Finally, we conclude with Chapter 5 by providing a concise summary of our work as well as the direction for future work.

Chapter 2

Background and Related Work

2.1 Distributed Computation

A distributed system is a system in which separate computers are connected on a network. Each component does its share of computation individually and then communicates the results via message-passing. The components are marked by concurrency, lack of a global clock, and independent failures of components [57]. Unlike parallel computing where all processors may have access to shared memory, distributed computing has components with its own private memory. Information is transmitted via message-passing mechanisms [29]. In recent years, machine learning community has come to embrace distributed computing [17]. The early work in this direction was launched by Zinkevich with his first implementation of a parallel stochastic gradient descent [66]. Since then, much more investigation has been conducted towards efficiently distributing ML [51, 30, 49].

2.2 Distributed Computation in Machine Learning

2.2.1 Stochastic Gradient Descent

Consider a supervised learning setup as such - given each example z that lives in $\mathbb{R}^d$ with its associated label y, we choose a family of functions $F, f_w(x)$ that is

parametrized by w. Our goal is to minimize the empirical loss function $l(f_w(x), y)$ averaged on the examples. A simple iterative process is proposed to find such a predictive function f.

$$w_{t+1} = w_t - \eta \frac{1}{n} \sum_{i=1}^{n} l(f(x_i, y_i)$$

Each iteration updates the weights w towards the minima and η is appropriately chosen as the learning rate.

Often, the number of examples z in each iteration is too large and the above method is not computationally feasible. This is where Stochastic Gradient Descent is useful [50, 8]. As a simplification of equation 1, we consider the gradient update from a stochastically chosen sample z_i

Algorithm 1: Stochastic Gradient Descent

```
Initialize random weight matrix W and learning rate η ;
while not at an appropriate minima do
    Randomly shuffle examples in the training set;
    for i=1,..n do
        w_i ← w_i − η∂_w Q(w_i);
    end
end
```

2.2.2 Parallelizing Synchronous-SGD

To better leverage the power of distributed systems, Sync-SGD is proposed with the following modification [66].

$$w_{i,t} \leftarrow w_{i,t-1} - \eta \partial_w c^i(w_{i,t-1})$$

Where each worker W_i applies the above equation individually, and then aggregates the resulting gradients, followed by averaging and applying those gradients.Aggregate from all workers $\frac{1}{k} \sum_{i=1}^{k} w_{i,t}$ and return v

This process translates to the following protocol below.

1. Distribute multiple copies of the training script and data where each of the k worker -

   (a) Independently sample data D_i

   (b) Forward and back-propagate the samples D_i through their model Θ_i

   (c) Computes gradient updates G_i

2. Average gradient updates G_i as $\sum_{i=1}^{k} G_i$

3. Apply the updates to model as $M_i \leftarrow \Theta_i + \sum_{i=1}^{k} G_i$

Further, researchers have been investigating the convergence behavior, performance, as well as stability of parallel-SGD methods [34, 64, 33].

2.2.3 Aggregation using Parameter-Servers

One could imagine several architectural designs that would allow for the achieve Sync-SGD as described above. The simplest such choice is of a single `Parameter-Server` [31, 12, 22].

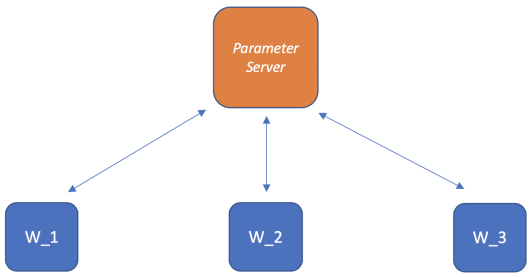

Figure 2-1: Single Parameter-Server Sync-SGD

While the single parameter-server approach works, it is not ideal from communications standpoint. A single aggregation point quickly becomes a network bottleneck as the user scales the number of workers. To alleviate this bottleneck, multiple

parameter-servers can be used. This parameter-server design is primarily used as the Tensorflow's out of the box distribution methods.

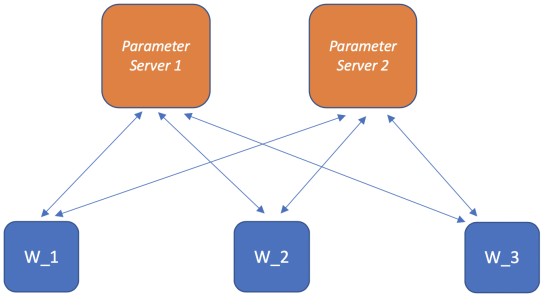

Figure 2-2: Multiple Parameter-Server Sync-SGD

2.3 Distributed Machine Learning Framework

2.3.1 TASO: Optimizing Deep Learning Computation with Automatic Generation of Graph Substitutions

Modern Machine Learning frameworks commonly utilize an abstraction known as a computation graph [36]. A computation graph is a representation of a computation where a `node` knows how to compute its value and its derivative w.r.t. each edge and the edges correspond to a function argument. Computation graphs are directed and often acyclic. In frameworks like Tensorflow, a computation graph is built first and then executed only when necessary. This approach, known as lazy evaluation, serves to reduce computation load and increase efficiency. Additionally, it allows for optimization of the computation graph itself. Consider two computation graphs as $A \times (B \times C)$ vs $(A \times B) \times C$, where the operator `matrix multiplication` is commutative. The results of the two computation graphs are equivalent, however, the runtime may differ. In practice, complex computation graphs can often be simplified to save runtime. Frameworks like Tensorflow apply graph substitutions to achieve this optimization.

TASO automates the optimization of computation graphs traditionally done manually by experts [26]. This approach is more scalable in the face of growing number DNN operators [7]. TASO achieves this by a cost-based backtracking search and then applying the optimization to the computation graph.

2.3.2 FlexFlow

Going beyond the standard data and model parallel approaches, FlexFlow introduces a richer parallelization space called SOAP - Sample, Operation, Attribute, and Parameter [37]. Sample and parameter dimensions refer to how training samples and model parameters are distributed. The attribute dimension is for different attributes within a sample. And operation dimension refers to how different operations are parallelized. FlexFlow introduces a deep learning framework that searches this SOAP space to produce optimal parallelization strategies. Given the larger search space, FlexFlow utilizes a more efficient evaluation method to quickly find an optimized strategy. They also contribute towards faster simulations of DNN model and show that the simulation can be run up to three times faster than the actual computation. The results show that FlexFlow can speed up throughput by up to 3.8x over other manual parallelization techniques.

2.3.3 PipeDream

Current parallelization techniques in deep learning employ intra-batch parallelization, in which a single iteration is split across workers and the gradients are then averaged [10, 13]. PipeDream introduces inter-batch parallelism to further improve throughput and reduce communication overhead [44]. Contrary to the traditional *model parallel* systems, PipeDream pipelines minibatch processing where different workers are computing different parts of the input at any instant of time. This ensures high utilization of GPUs and low communication overhead. They achieve up to 5x speed-up in time to target accuracy for experiments with five different DNNs.

However, their system does not speed up RNNs [40] as much due to the inherent challenges with recurrent neural networks.

2.3.4 Horovod: Fast and Easy Distributed Deep Learning in Tensorflow

Horovod is Uber's open-source distributed machine learning framework and is the most widely used and well-maintained framework in this field [52]. Horovod implements AllReduce using Open-MPI (message-passing interface) allowing for fast and easy distributed training of ML models.

Message Passing Interface

Message Passing Interface (MPI) is a standard communication interface for distributed programs [16]. MPI's abstractions consist of a `communicator`, which is defined as the group of processes each assigned a unique `rank`. Processes refer to each other using their ranks. MPI defines communication of one-to-one, one-to-many, and many-to-many paradigms. This fits naturally into the distributed workers' operations since two workers may need to communicate directly with each other, or one process may need to send/receive information from multiple processes, and, finally, multiple processes may communicate with several processes at once [57].

MPI Send Directive

```
MPI_Send(
    void* data,
    int count,
    MPI_Datatype dtype,
    int destination,
    int tag,
    MPI_Comm communicator)
```

MPI Send Directive

```
MPI_Recv(
    void* data,
    int count,
    MPI_Datatype dtype,
    int source,
    int tag,
    MPI_Comm communicator,
    MPI_Status* status)
```

AllReduce

Reduce MPI defines many operators as parts of its collective operations library. One of the most important ones is Reduce. Reduce is a concept derived from functional programming [58, 45]. Formally, we define D a *Collection* of objects d_i and a associative reduction operator F such that it can reduce D to a singular value x. For example, consider the binary function `max(a,b)` and input `[1,-10,13,2,20]`. Reduce operator applied to this input given the function would yield 20.

MPI Reduce Directive

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

Often, we would like to access reduced results across multiple parallel processes

rather than just the root process. This directive is defined as the `AllReduce` [6]. Here, processes reduce results across workers and then broadcast across all workers. Thus, an `AllReduce` is a `Reduce` operator followed by a `Broadcast` operator. See Figure 2-3.

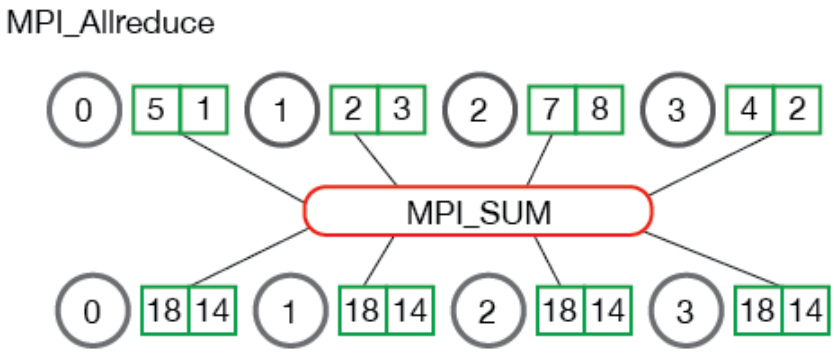

Figure 2-3: MPI AllReduce example with MPI_SUM

MPI AllReduce Directive

```
MPI_Allreduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    MPI_Comm communicator)
```

Ring-AllReduce In the early versions of All-Reduce algorithms, a centralized *parameter-server* was required. This server is responsible for coordinating the messages between workers. Researchers quickly realized the inherent limitations of this approach - it does not scale. Tensorflow has parameter-server based distribution strategy built-in but it suffers the same drawbacks of lacking scalability [1].

Ring-AllReduce does away with needing a centralized parameter-server. As the name suggests, in this algorithm workers are connected in a ring topology. Each of

the N workers exchanges information with its two neighbors. One can show that after $2 * (N - 1)$ messages, all workers become synchronized with the same values. This algorithm is bandwidth-optimal according to Baidu's paper [46]. Figure 2-4 shown an example of workers exchanging updates using Ring-AllReduce.

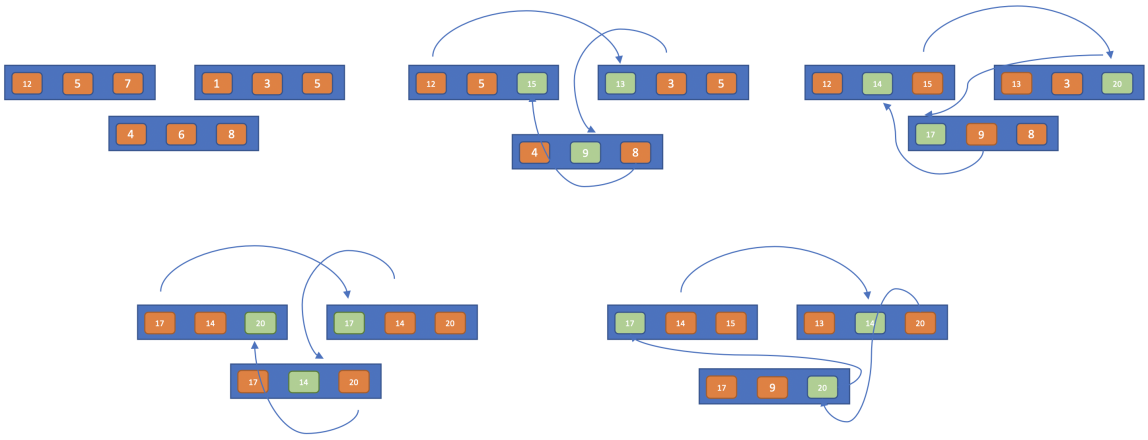

Figure 2-4: Decentralized Ring AllReduce Sync-SGD

Horovod As a system, Horovod combines all the ideas discussed in this section so far. Therefore, Horovod's contributions include -

- Standalone Python package that works with Tensorflow Keras backend.

- Leveraging NCCL All-Reduce algorithm as an optimized GPU implementation of Baidu's Ring-AllReduce.

- APIs to enforce consistent initialization of models across workers.

Usage Horovod provides high-level APIs that ensure that the user is able to correctly and efficiently distribute their machine learning training.

Listing 2.1: Horovod usage example

```
import tensorflow as tf
import horovod.tensorflow.keras as hvd

# Initialize Horovod
hvd.init()

# Pin GPU to be used to process local rank (one GPU per process)
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
if gpus:
    tf.config.experimental.set_visible_devices(gpus[hvd.local_rank()], 'GPU')

# Build model and dataset
dataset = ...
model = ...
opt = tf.optimizers.Adam(0.001 * hvd.size())

# Horovod: add Horovod DistributedOptimizer.
opt = hvd.DistributedOptimizer(opt)


mnist_model.compile(loss=tf.losses.SparseCategoricalCrossentropy(),
                    optimizer=opt,
                    metrics=['accuracy'],
                    experimental_run_tf_function=False)

callbacks = [
    # Horovod: broadcast initial variable states from rank 0 to all other processes.
    # This is necessary to ensure consistent initialization of all workers when
    # training is started with random weights or restored from a checkpoint.
    hvd.callbacks.BroadcastGlobalVariablesCallback(0),
]

model.fit(dataset,
          steps_per_epoch=500 // hvd.size(),
          callbacks=callbacks,
          epochs=24,
          verbose=1 if hvd.rank() == 0 else 0)
```

2.3.5 KungFu

Horovod, being the first reliable distributed ML framework, has gained popularity in the industry. However, Horovod has its own limitations [52]. First, Horovod's distribution strategy is static. This means the user can't change the workers dynamically to adapt to their amount of workload and compute capacity. KungFu aims to provide an adaptive distributed framework [38] by allowing the training to dynamically adjust the number of workers participating in a training job. Also it rewrites the communications layer instead of using `MPI` and thus allowing for faster syncing across the workers. KungFu introduces parallel `AllReduce` operations. This is made possible by associating a unique hash-key corresponding to each tensor and then buffering the data in its queue before reducing. That said, KungFu still has a lot of room for improvement. In section 4.5 we describe our lessons learned working with KungFu. While we relied on KungFu to implement our ideas in this thesis, our recommendation to the reader is to use Horovod instead of KungFu since we encountered several unexplained challenges and bugs in KungFu.

KungFu focuses on adaptive scaling of resources while training large machine learning models. It provides basic distribution API similar to that of Horovod, but it also adds new methods such as - `resize_cluster` and `monitor`.

Listing 2.2: KungFu base API usage example

```
import tensorflow as tf
from kungfu.tensorflow.optimizers import SynchronousSGDOptimizer
from kungfu.tensorflow.initializer import BroadcastGlobalVariablesOp

# Build model...
loss = ...
opt = tf.train.AdamOptimizer(0.01)

# KungFu Step 1: Wrap tf.optimizer in KungFu optimizers

opt = SynchronousSGDOptimizer(opt)

# Make training operation
train_step = opt.minimize(loss)

# Train your model
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    # KungFu Step 2: ensure distributed workers start with consistent states
    sess.run(BroadcastGlobalVariablesOp())

    for step in range(10):
        sess.run(train_step)
```

KungFu's Design Primitives

This section contains important background of a select few design primitives from KungFu that are used in AIKIDO.

- `Reduce graph`: A graph $\mathcal{G}(V, E)$, where $v \in V$ represents a worker and $e \in E$ is an edge (v_1, v_2) representing direction of a `Reduce` primitive from v_1 to v_2
- `Broadcast graph`: A graph $\mathcal{G}(V, E)$, where $v \in V$ represents a worker and $e \in E$ is an edge (v_1, v_2) representing direction of a `Broadcast` primitive from v_1 to v_2
- `Strategy`: A `Strategy` is defined as a pair of `Reduce graph` and `Broadcast graph`. Formally a `Strategy` $S = \{\mathcal{G}_{\mathcal{R}}(V, E), \mathcal{G}_{\mathcal{B}}(V, E)\}$

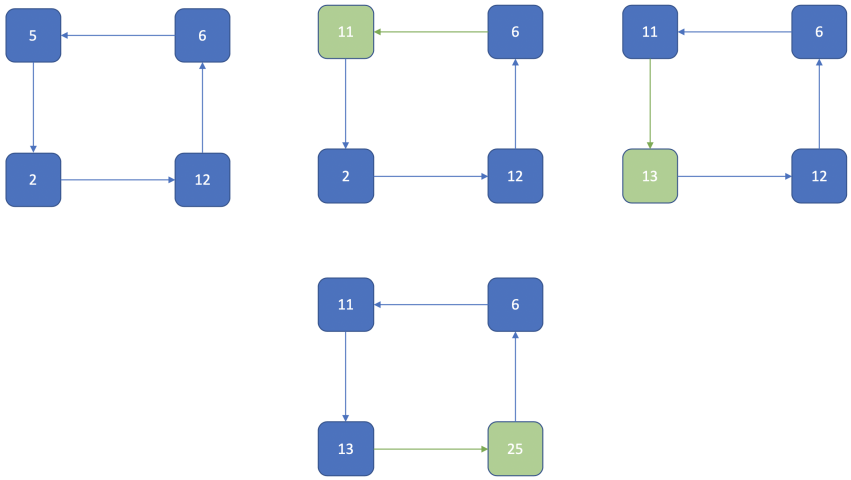

Figure 2-5: Ring Strategy: Reduce graph

Let us consider how these strategies are executed. First, we look at the often used ring strategy, which is based on the popular `Ring-AllReduce` algorithm. Figure 2-5 and 2-6 respectively illustrate the `Reduce graph` and `Broadcast graph` topologies.

Reduce Graph

1. Each of the four workers W_i where $i \in \{1, 2, 3, 4\}$, begin in their initial state holding data D_i in memory.
2. W_1 sends D_1 to W_2 which sums it to D_2
3. W_2 sends $D_1 + D_2$ to W_3 which sums it to D_3
4. W_3 sends $D_1 + D_2 + D_3$ to W_4 which sums it to D_4

Broadcast Graph

1. The worker W_4 begins in its initial state holding data $\sum_{n=1}^{4} D_i$ in memory.
2. W_4 sends $\sum_{n=1}^{4} D_i$ to W_1 which replaces the existing buffer.
3. W_1 sends $\sum_{n=1}^{4} D_i$ to W_2 which replaces the existing buffer.
4. W_2 sends $\sum_{n=1}^{4} D_i$ to W_3 which replaces the existing buffer.

Next, we consider the *Star* strategy, based on the topology of a star graph structure. Figure 2-7 illustrates the initial state and the following `Reduce graph` and `Broadcast graph` for a 4 worker *Star* Strategy.

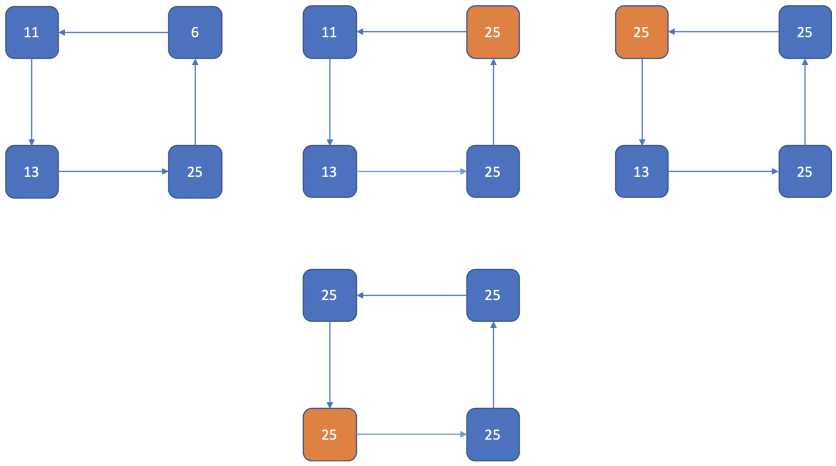

Figure 2-6: Ring Strategy: Broadcast Graph

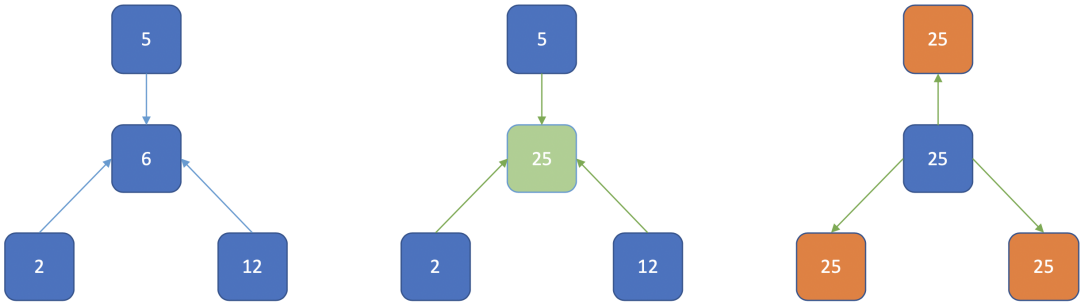

Figure 2-7: Star Strategy: *left to right* Topology with state of workers, Reduce graph ops, Broadcast graph ops

1. Each of the four workers W_i where $i \in \{1, 2, 3, 4\}$, begin in their initial state holding data D_i in memory.

2. **Reduce Step:** The workers W_2, W_3, W_4 all send their data to W_1, which sums it up.

3. **Broadcast Step**: The worker W_1 stores $\sum_{n=1}^{4} D_i$ in memory, which it sends to workers W_2, W_3, W_4, each of which replace their existing buffer with $\sum_{n=1}^{4} D_i$

2.4 Approaches to Straggler Protection

Current approaches to straggler protection largely fall into two camps. One idea is to use the synchronous training combined with replication and intelligent restarting of computation [48, 63]. Next, we have asynchronous training as a means to avoid the straggler problem altogether [35, 62, 3]. While the recent progress in async is exciting, synchronous algorithms are often preferred due to better performance and convergence guarantees.

2.4.1 Gradient Coding

Gradient coding turns to the area of coding theory to provide better strategies for replicating data blocks that are tolerant to failures and stragglers [48]. To understand their basic idea, consider three workers labeled as W_1, W_2, W_3 each computing gradients in for the synchronized SGD algorithm. Without modification, the workers would compute gradients G_1, G_2, G_3. If one of the workers W_i is a straggler and loses G_i, the aggregation step can not be completed. However, now consider a slightly modified gradient sharing scheme as follows. The workers $W_1, W_2, W3$ now respectively compute gradients as $G_1/2 + tG_2, G_2 - G_3, G_1/2 + G_3$. Now, the vector $G_1 + G_2 + G_3$ is in the span of any of the two vectors out of three. The major contribution from this paper is that this gradient coding approach does not require feedback and coordination, and that we can configure each worker to independently send linear combinations of gradients such that the aggregated sum can be obtained from the combinations [48]. This work shows the promise of mining the rich field of coding and information theory for developing better systems.

2.4.2 Google's Sync-SGD Primary-Backup System

Google's Revisiting Distributed Synchronous SGD, hereafter referred to as **GRD-SGD**, presents the case for the need to mitigate stragglers in distributed ML training [9]. First, they demonstrate the straggler effect in aggregating increasing number of gradients from distributed workers. Their results closely follow our proposed model for stragglers in section 3.1.1 and chart a poisson CDF. Next, GRD-SGD proposes a straggler-mitigating system design based on the parameter-server paradigm. In their method, the parameter-server P_i chooses to drop the last b number of updates from their corresponding delayed workers. Following this approach, GRD-SGD is able to reach both faster and better convergence [9].

There are many aspects of GRD-SGD that could be improved in future work, but their fundamental bottleneck comes from having to rely on centralized parameter-server architecture [9]. While parameter-servers make it easier to perform centralized logic such as that of dropping the tail-end of gradient updates during aggregation, this also inherently limits the scalability of such a system.

2.4.3 Async Decentralized Parallel Stochastic Gradient Descent

Most of the techniques commonly used in distributed machine learning are either 1) synchronous 2) centralized asynchronous. Synchronous algorithms suffer from the issue of stragglers and therefore don't perform well in heterogeneous or communication-bottlenecked environment. Asynchronous algorithms aim to solve this problem by making SGD async. However, often this achieved by using a centralized parameter-server [31]. This leads to scaling issues as parameter-server can get overwhelmed in the face of larger number of workers. AD-PSGD provides a *parallel* and *decentralized* algorithm for Stochastic Gradient Descent along with convergence bound of $\mathcal{O}(\frac{1}{\sqrt{K}})$

achieving linear speedup w.r.t. number of workers [35].

An interesting aspect of AD-PSGD is their design to avoid deadlocks that uses bipartite graph topology. This ensures that the graph is partitioned into active set(A) and passive set(B), where the edges only go from one set to the other [35].

2.4.4 Trend-Smooth

Asynchronous SGD algorithms are an active area of research. Currently, the major issue with such algorithms is that they 1) converge slowly due to stale information and 2) often don't converge on global minima [21, 65]. Trend-smooth paper aims to accelerate asynchronous SGD by smoothing parameters, i.e, shrinking the learning rate in some dimensions where the gradient directions are opposite of the parameter trend [12]. Trend-Smooth is empirically shown to asynchronously converge to state of the art accuracies in MNSIT and CIFAR-10 datasets [11].

Chapter 3

AIKIDO: A First Step Towards Straggler-Mitigating Distributed Machine Learning Framework

As Machine Learning (ML) becomes ubiquitous and training models increasingly harder, there is a greater need of an easy-to-use, flexible, straggler-resistant distributed ML framework. We propose our framework, AIKIDO. AIKIDO's goal is to curb the effects of stragglers in large-scale training jobs with Ring-AllReduce communication pattern. In synchronous distributed computation, stragglers are workers that lag behind the rest of the workers. This section describes our proposed framework to help mitigate the effects of stragglers for distributed machine learning.

Straggler nodes cause delays in synchronous computation. This happens when the results from all workers need to be aggregated before proceeding to next state in the computation. The problem of stragglers is more pronounced in the case of distributed machine learning due to iterative nature of Stochastic Gradient Descent like algorithms. Thus, even a few stragglers every iteration can significantly slow down performance as measured via throughput and time to accuracy.

Stragglers have always posed a challenge in the field of distributed computation [28, 59, 15, 19]. They waste precious computation cycles that need to be recovered either by replicating machines or restarting the computation. In ML, the problem is further exacerbated due to the iterative and synchronous nature of the underlying training algorithms.

State of the art methods for straggler mitigation in distributed computation rely on replication which can be a costly measure as it requires additional resources [59]. Prior work proposed straggler mitigation via a primary-backup architecture [9]. We refer to this approach as GRD-SGD. However, this architecture relies on the centralized parameter-servers and hence suffers from the same scaling limitations that are imposed by parameter-servers [5]. Despite frameworks like Horovod gaining recent popularity, no current frameworks provide out-of-the-box straggler mitigation mechanisms for ML. Table 3 below summarizes the state-of-the-art training frameworks and compares them with AIKIDO.

Approach	Aggregation	Open-Source	Straggler Mitigation
GRD-SGD [9]	Parameter-Server	No	Yes
Horovod [52]	Ring-AllReduce	Yes	No
AIKIDO	**Ring-AllReduce**	**Yes***	**Yes**

Table 3.1: State of current distributed ML frameworks and where AIKIDO fits. * Our code is available online at `https://github.com/ayushs7752/AIKIDO`

In proposing AIKIDO, we hope to fill the existing gap in curbing the effects of stragglers in large-scale distributed ML. We build on two primary insights. First, modern neural networks and training methods are fairly robust against occasional loss of data. Second, parameter aggregation can be performed in a decentralized fashion; that is without needing parameter-servers due to existence of algorithms like Ring-AllReduce. We combine these two ideas and put forward AIKIDO where we design a primary-backup architecture that curbs stragglers by selectively removing them from the AllReduce topology while retaining the fully-decentralized nature of underlying Ring-AllReduce algorithm.

Our work in developing AIKIDO makes the following contributions

- A Ring-AllReduce based primary-backup architecture including a skip and active backup strategies to mitigate stragglers.
- A *Ring-Reshape Module* that helps in curbing stragglers at the aggregation level of training iterations.
- An easy-to-use straggler simulation framework for quantifying and experimenting with stragglers in real-world training.

3.1 Straggler Mitigation within a Ring-AllReduce Aggregation Paradigm

AIKIDO is built on top of KungFu [38] which provides us with distributed computation primitives such as *Reduce*, *Broadcast*, and *Sync-SGD* as described in Chapter 2.

AIKIDO resides in the shared context that is common across the N workers connected in a directed graph topology. Underneath AIKIDO, we reuse KungFu's communications layer for message-passing. AIKIDO initializes the workers in a canonical ring topology and changes that topology according to a pre-determined configuration

file that mocks the straggler behaviour. In real world, this configuration file should be replaced by a straggler detection daemon that runs as a background process inside each of the W_i workers AIKIDO's context. The daemon should periodically monitor two key metrics: the current iteration time t_i and the current waiting time $wait_i$ for a given gradient update from the worker W_{i-1}. While this thesis does not implement the straggler detection daemon, we propose two potential methods to implement it:

- Keep the running average of waiting times $\sum_{i=}^{T} wait_i$. Flag the previous worker as a straggler if $\frac{wait_i}{\sum_{i=}^{T} wait_i} > wait\_threshold$. Here, $wait\_threshold$ is an adjustable parameter that the user can specify for their use-case.

- Keep the running average of waiting times $\sum_{i=}^{T} wait_i$ as well as iteration times $\sum_{i=}^{T} t_i$. Flag the previous worker as straggler if $\frac{t_i}{\sum_{i=}^{T} t_i} > iteration\_threshold$. Here, $iteration\_threshold$ is a parameter that would differ in each application and therefore would be chosen by the user per their constraints.

We note that for this thesis, we did not realize any of these methods and simply mock out the straggler daemon component with a configuration file. The configuration file includes a worker ID, iteration number, and the amount of straggling time. AIKIDO reads this configuration file and hard-codes the delays accordingly by adding a sleep before the reduce operation. Mocking straggler behavior lets us better control our experiments and we leave implementing the straggler detection daemon to future work.

Straggler Simulator in AIKIDO

Our straggler simulator component in AIKIDO consists of the following parts.

- Straggler Model: Formalizing the concept of stragglers for our project as well as theoretical justification for our simulated straggler model.

- DelayConfig: Pipeline for generating straggler configurations and the parameters we can vary.

- DelayOp: Implemented operator for injecting delays into a given worker W_i.

We describe the straggler model here in detail, followed by a summary of the DelayConfig and DelayOp.

Straggler Model: Let us build a statistical model for the arrival times of gradient updates in distributed machine learning. Here are the key assumptions we will make.

- The arrival of gradient from worker W_i is independent of worker $W_j \forall j \in (1, k)$ if $j \neq i$
- The number of arrivals in a continuous interval takes discrete values $k = 0, 1, 2, ..$

Given the above assumptions, a natural candidate for the gradient arrival times is the Poisson Distribution. Let k be the number of gradient arrivals in any interval T and λ be the rate parameter. Then we have $f(k; \lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$. An important consequence of a Poisson distribution for gradient updates is that while most workers finish in average time, there is a tail-end of updates that lag behind.

DelayConfig and DelayOp

We utilize *DelayConfig* and *DelayOp* to configure the straggler simulation as well as to inject the given amount of delay in an arbitrary worker during an iteration. The *DelayConfig* is a file generated using the above Poisson model of straggler rate distribution. Each row consists of the `WorkerID`, `IterationID`, and `Delay`. Next, the *DelayOp* takes the *DelayConfig* and inserts simulated time delay into the node `WorkerID` right before the reduce step, at the iteration `IterationID` for a duration of `Delay`.

Put together, this system, as depicted in Figure 3-1, allows us to introduce arbitrary rates of straggling in any number of distributed workers and see the aggregate

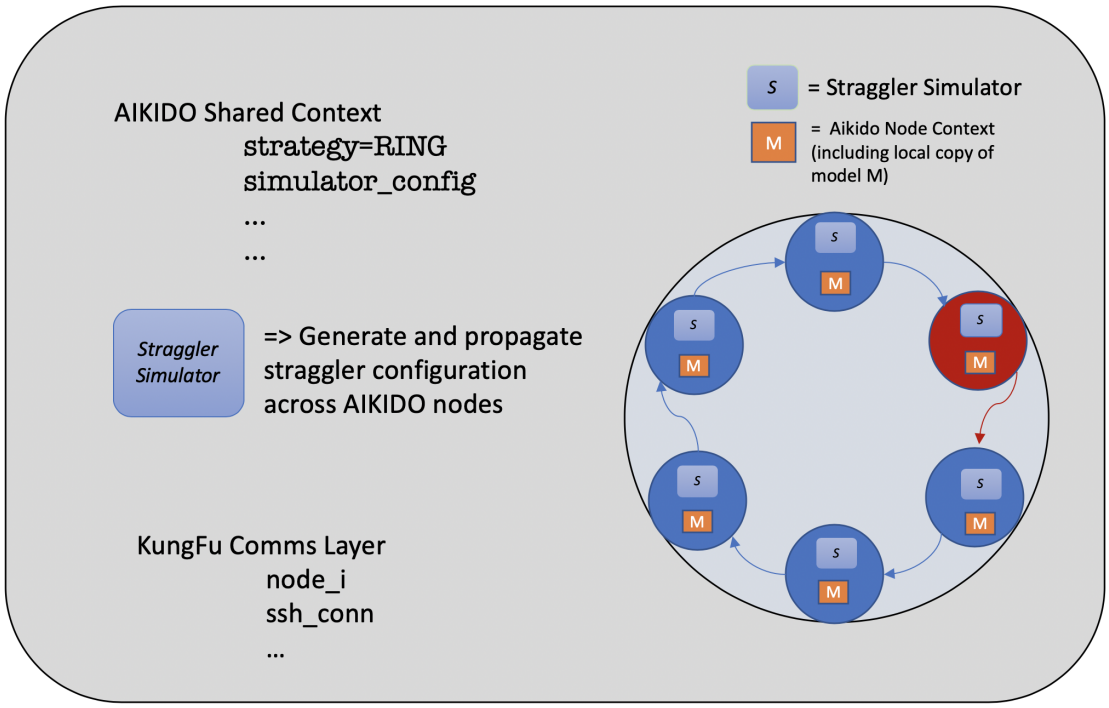

Figure 3-1: AIKIDO's Overview of Design with Straggler Simulator

behavior as reflected in mini-batch iteration times for a given model. The `Straggler Simulator, S` consists of the straggler model, delayConfig, and delayOp. Each of the W_i workers start out with the same delayConfig in their local storage. Once the workers start training a given model using Sync-SGD, the `Ring-Reshape Module, R` communicates with `S` to load the config, and reshape the topology to switch out the straggler worker S_t out of the reduce graph with a low operational overhead. Note that AIKIDO keeps the worker S_t connected in broadcast graph. This ensures that the straggler worker does not impact the iteration time of the rest of the ring during this given iteration, and received the latest updated copy of the model. The next section describes the `Ring-Reshape Module` in detail.

3.2 Flexible Topology for Collective Communication Operations

AIKIDO introduces the `Ring-Reshape Module` as one of our primary contributions. This module allows for seamless switching of `AllReduce` topologies dynamically during a run. This is critical because of two main reasons. First, our design does not rely on centralized parameter-servers. Secondly, straggler behavior is often sporadic and short-lived [9]. As discussed in our straggler model section, we assume that a Poisson distribution governs the nature of straggler finish times. The independence of straggler finish times implies that the phenomenon occurs often on the order of iterations - so few hundred milliseconds, not minutes. Existing distributed ML frameworks either do not address this problem or they turn to the assumption that stragglers persist over the duration of at least a few minutes. As such, they are fundamentally limited to this low level of granularity because adding and/or removing servers into a participatory topology takes time at least on the order of a few minutes. In contrast, AIKIDO's flexible topology is implemented at the level of abstraction of `Collective Communications Ops` and therefore it frees us from the low granularity constraint as communication ops are orders of magnitudes faster than rebooting a machine. Consequently, this approach puts our performance on par with centralized approaches like GRD-SGD [9].

In AIKIDO's *Skip* strategy, we designate straggler nodes using the straggler simulator and its delay config. Next, the *Ring-Reshape Module* seamlessly swaps out the straggler node and completes the ring topology with the next available worker. Additionally, we keep the straggler node in `Broadcast graph` so it can receive its updated copy of the ML model. This works for a few reasons. First, note that a straggler node, by definition, is the one lagging behind in its share of the distributed computation and

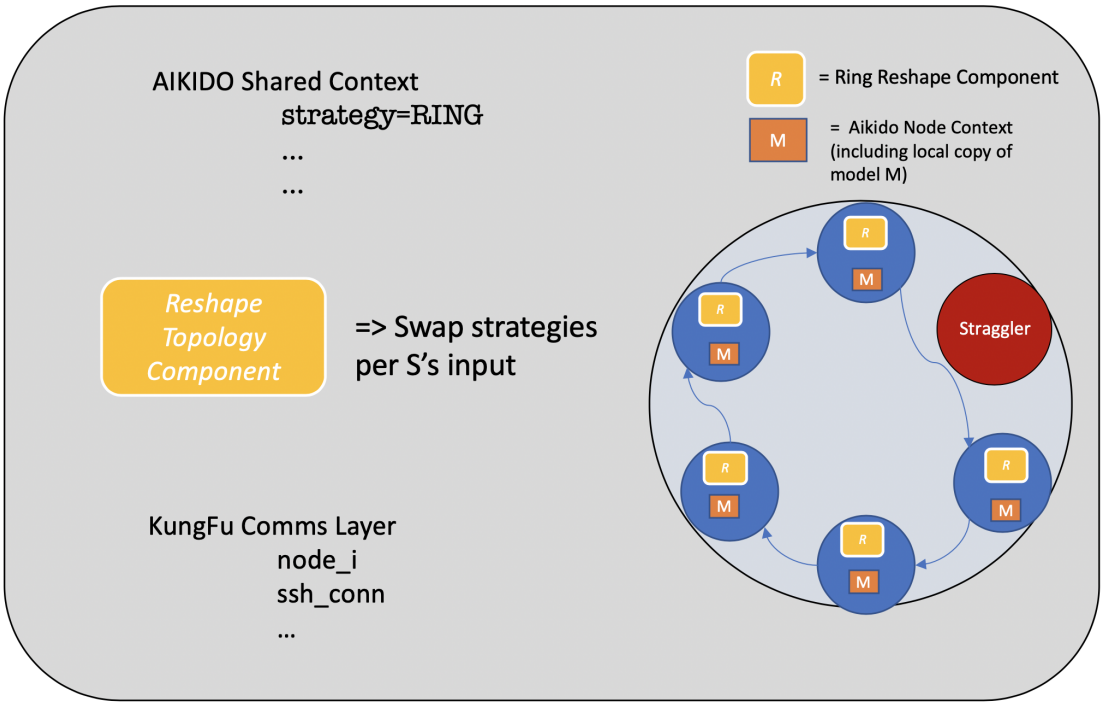

Figure 3-2: Ring-AllReduce Skip in AIKIDO

therefore does not return its gradients in time. Assume a simple yet characteristically typical scenario with a set of nodes and edges as $W_{i-2} \rightarrow W_{i-1} \rightarrow W_i$ with W_{i-1} being the straggler node. Here, we introduce our custom operator - `ReshapeStrategy`. The `ReshapeStrategy` operator, when called before the All-Reduce step in Sync-SGD during this iteration, changes the workers' topology from a ring to a ring with the following edge modified as $W_{i-2} \rightarrow W_i$. In effect, this accomplishes the task of bypassing the straggler node for the given iteration in which worker W_{i-1} straggles. The straggler simulator continuously runs and returns the delay configuration for each iteration and the *Ring-Reshape Module* updates the workers' topology to the right configuration.

We describe this mechanism using an illustrative example below as shown in Figure 3-3:

1. Initialize AIKIDO context with 4 workers in a ring topology running a sync-SGD iterations. Initialize straggler simulator with a given delay configuration.

2. The workers W_i begin in their initial state holding gradient updates data D_i in memory.

3. Straggler Simulator S delays W_4 for this iteration.

4. The Ring-Reshape Module removes W_4 out of the Ring-Reduce topology and instead connects W_1 with W_3 to exchange gradients updates and successfully finish this iteration.

5. Worker W_4 is kept in the Broadcast graph topology so it receives the updated copy of the model θ.

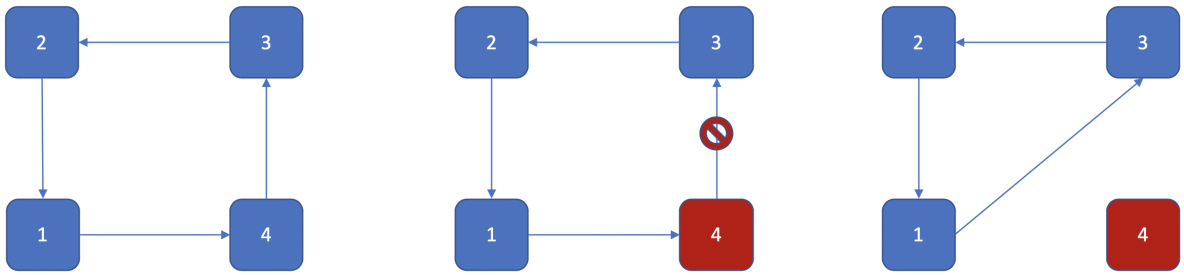

Figure 3-3: AIKIDO's Skip Mechanism Example

3.3 Active-Backups as an Alternative to Skip Strategy for Straggler Mitigation

Unlike *Skip-Backups* approach, we also consider backups which actively substitute for lost results. This form of replication is already prevalent in traditional distributed computation and it is worth exploring it for machine learning [32, 60, 57]. Similar to the *Skip-Backup* case, assume a typical scenario with a set of nodes and edges as $W_{i-2} \rightarrow W_{i-1} \rightarrow W_i$ with W_{i-1} being the straggler node. Here, we use our custom operator - `ReshapeStrategy` but with the additional introduction of active-backup A_i. `ReshapeStrategy` operator, when called before the AllReduce step in Sync-SGD during this iteration, changes the workers' topology from a ring to a ring with the following edge modified as $W_{i-2} \rightarrow A_i \rightarrow Wi$. AIKIDO keeps the set of active backups A_i connected in the Broadcast graph so they share the same updated copy of the model as the rest of the primary workers W_i. Note that primary workers are defined as workers that participate in the ring in all iterations unless a given primary straggles, in which case an active-backup is swapped in lieu of that primary worker.

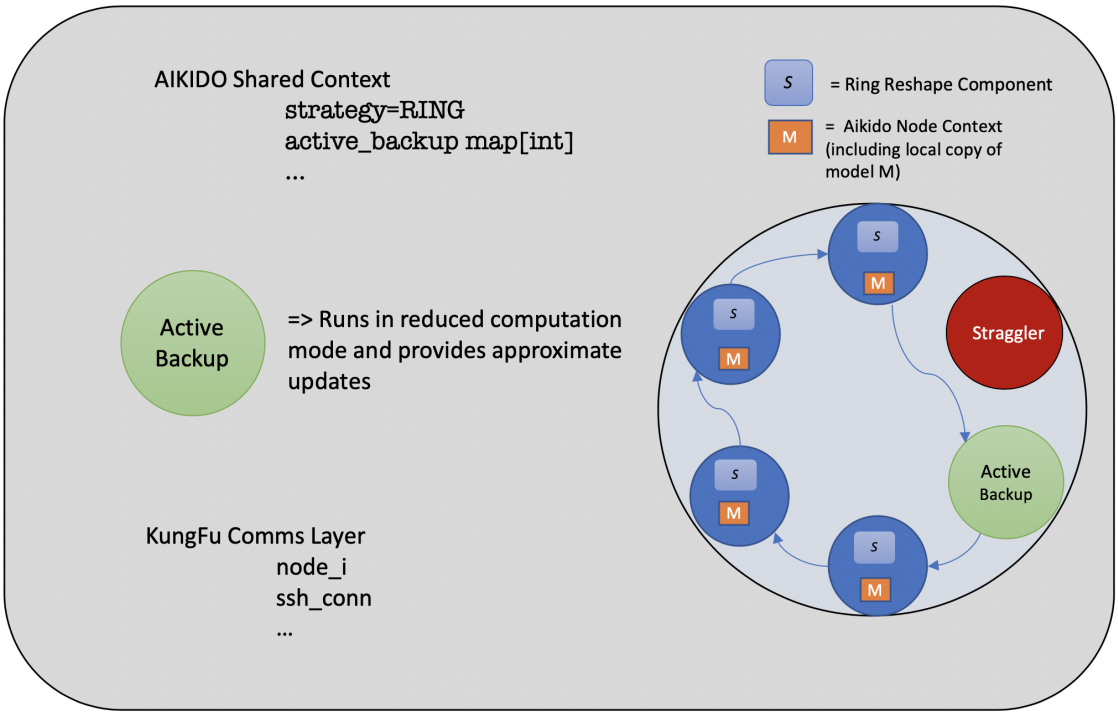

Figure 3-4: Active Backup in AIKIDO

Figure 3-5 provides an example for a case with four workers:

1. Initialize AIKIDO context with 3 primary workers in a ring topology running sync-SGD. Initialize straggler simulator with a given delay configuration. Finally, initialize an active backup worker A_i.

2. The workers W_i begin in their initial state holding gradient updates data D_i in memory.

3. Straggler Simulator S delays W_4 for this iteration.

4. The Ring-Reshape Module removes W_3 out of the Ring-Reduce topology and instead connects W_1 to $A_i to$ W_2 to exchange gradients updates and successfully finish this iteration.

5. Worker W_3 is kept in the broadcast graph topology so it receives the updated copy of the model θ.

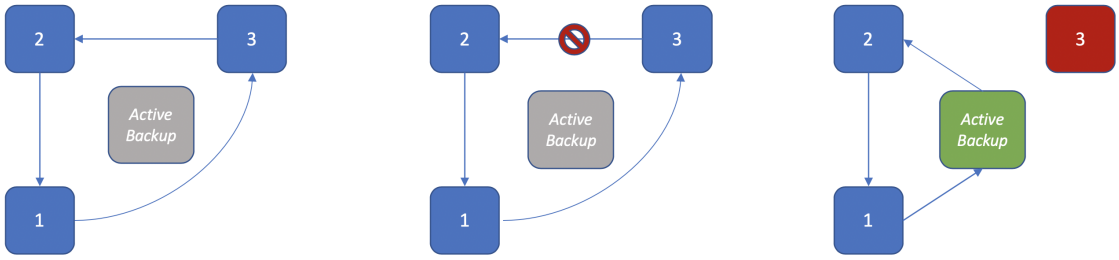

Figure 3-5: Ring-AllReduce Active Backup Architecture Example

In this thesis, we explore active backups with random partitions of data. Modifications to active backups could be further studied to establish the benefits of this approach.

3.4 The AIKIDO Profiler

As we worked on this project, we often found ourselves wanting more capable tools for analysing distributed ML. As a result, we implement a profiling tool to help with debugging. There is large variation among different models in terms of their compute/communications profile and discovering it quickly would help users effectively target their optimization efforts. Additionally, being able to visually glean what the time distribution is between all the complicated and interacting steps in a distributed ML job would be crucial to advance the state of research in this area. This is where a *Profiler* is useful.

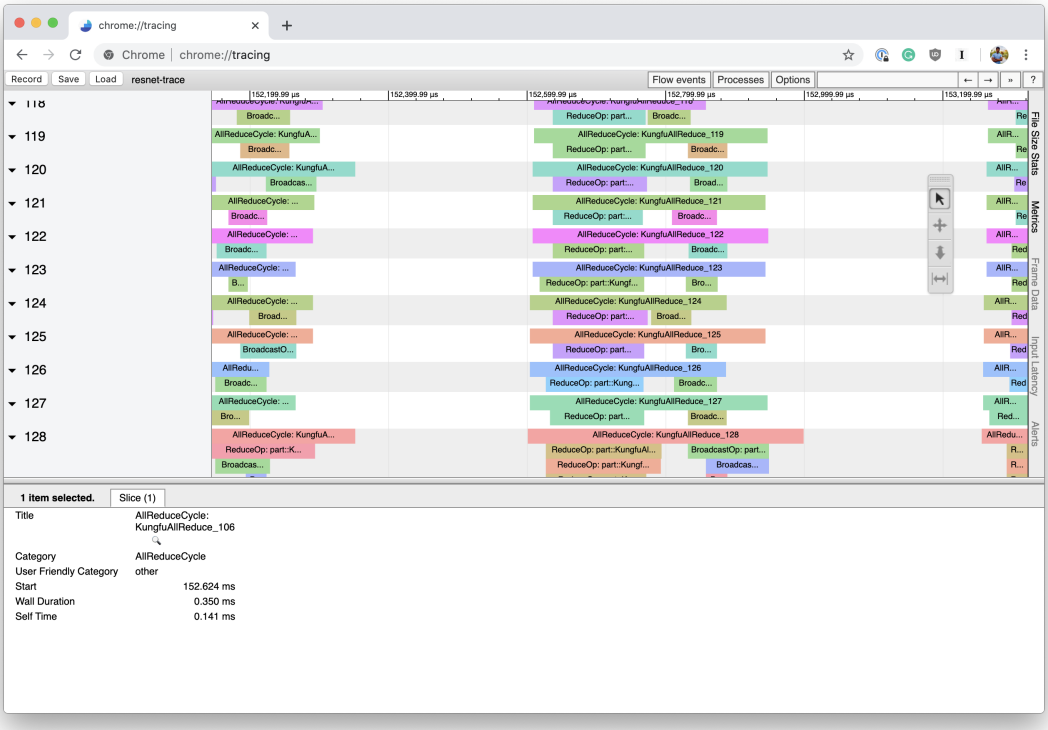

Figure 3-6: AIKIDO's Profiler in Chrome

AIKIDO*'s* profiler is composed of two components -

1. **The AIKIDO Logger:** This component provides an API to enable logging of the key steps within an all-reduce cycle such as beginning of the allreduce operation, beginning of reduce, end of reduce operation, beginning of broadcast operation, and end of broadcast operation.

2. **Chrome tracing for visualization**: Google Chrome provides a convenient utility within the browser to visualize traces [54]. We utiliz Google Chrome's tracing API and therefore build AIKIDO logger to be compatible with Chrome's tracing JSON format.

Chapter 4

Experiments and Results

In this section, we evaluate the performance of AIKIDO on ResNet-50 with synthetic training data, Cats vs. Dogs dataset, and ImageNet. For synthetic benchmarks and Cats vs. Dogs, we use a local batch size of 128 images. For ImageNet, we use TensorFlow's guidelines on loading and processing ImageNet data on local storage in TFRecords format [55]. For ImageNet, we vary the local batch-size between 32 and 64, which we specify for any given experiment below. To evaluate AIKIDO's performance, we choose Google Cloud Engine and provision virtual machines with four NVIDIA Tesla V100 GPUs per virtual machine. We scale the number of GPUs between 4-96 for various experiments, with exact parameters specified below for each experiment. As noted earlier in Chapter 3, we base AIKIDO's implementation on top of KungFu [38].

For our experiments to compare ideal, straggler, skip, and active backup runs, we simulate a 20% straggler rate. This is done as follows: AIKIDO adds 75ms of delay to one of the workers on top of the 180ms iteration time, every alternate iteration. On average, this amounts to $\frac{\frac{1}{2}*75*100}{180} = 20\%$ delay. Further, note that we define ideal performance to have linearly scaled throughput. In our setup, ResNet-50 ideal iteration time with local batch size 32 on ImageNet with 4 GPUs is observed at around

130ms on average. With 16 GPUs, the same configuration yields ideal iteration time at around 175ms. Note that this is due to imperfect scaling of existing distributed ML framework.

4.1 Current performance measures in distributed ML

One of the first results we establish in this project is to quantify the gap between linear scaling and measured scaling of distributed ML as the number of workers scale. This is necessary to get a measure of the room of improvement present in the current system.

4.1.1 Measuring scalability of GPU throughput via ResNet-50 Benchmark

In this experiment, we measure the performance in terms of GPU throughput (images/sec) in ResNet-50 Sync-SGD with synthetic training data [55]. We scale the number of GPUs from 4 to 56. Figure 4-1 shows the average aggregate throughput of GPUs for each experiment. We note that despite running each GPU close to the maximum capacity, the throughput does not scale linearly as compared to the linear speedup line. In fact, scaling up from 4 GPUs to 56 GPUs only achieves $\frac{1}{2}$ of the linear average aggregate throughput.

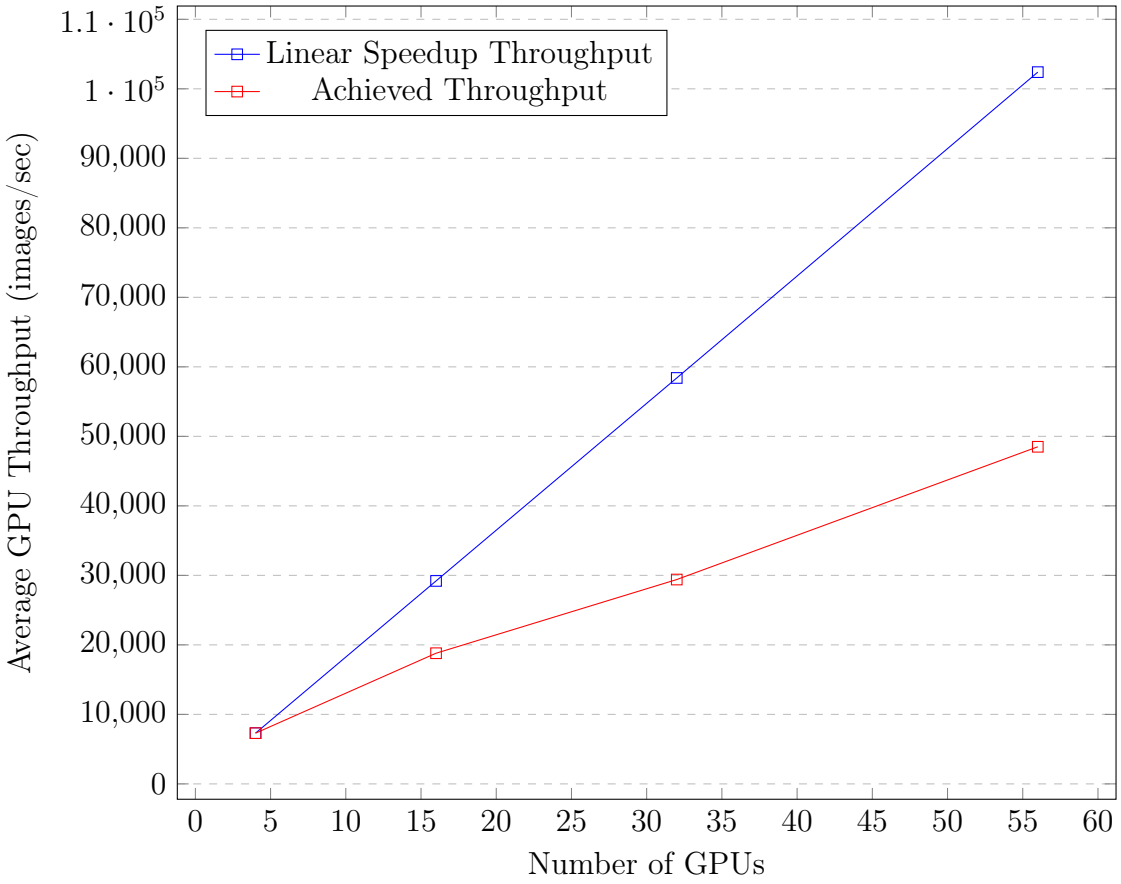

Figure 4-1: ResNet-50, benchmark – Linear speedup vs. Achieved throughput

4.1.2 Measuring scalability of iteration times via ResNet-50 on Cats vs. Dogs dataset

Next, we conduct a similar experiment to measure time per epoch of ResNet-50 on the popular Cats vs. Dogs dataset [27]. This dataset consists of 25,000 photos of dogs and cats. In this run, we tested the number of GPUs in the range of 4-56. Figure 4-2 shows a significant gap in performance as the achieved average time per epoch is 3.8× slower than the ideal linear speedup time per epoch. As mentioned earlier, linear speedup performance is calculated by linearly extrapolating the time per epoch of one VM.

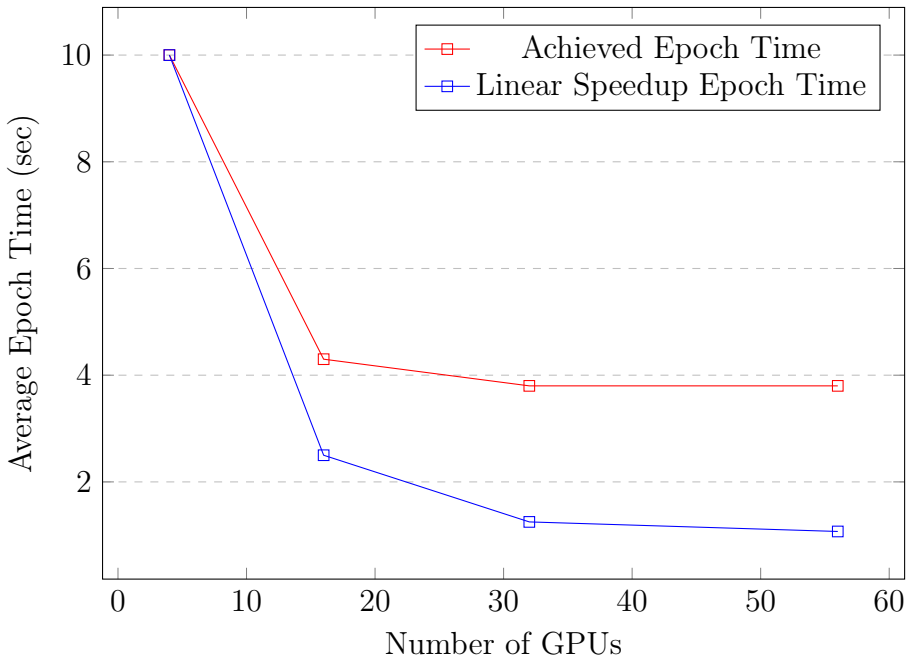

Figure 4-2: ResNet-50, Cats vs Dogs dataset - Achieved vs Linear epoch time

4.1.3 ResNet-50 on ImageNet: Training to Convergence

To establish full baseline performance on a state-of-the-art model, we trained ResNet-50 on ImageNet with local batch size of 128 for 90 epochs until convergence to achieve 92% top-5 accuracy. Figure 4-3 shows that the total time to train, as we double the compute and go from 48 GPUs to 96 GPUs, decreases only a factor of 1.5× and that there is an extra lag of 50 minutes between ideal and achieved time to convergence.

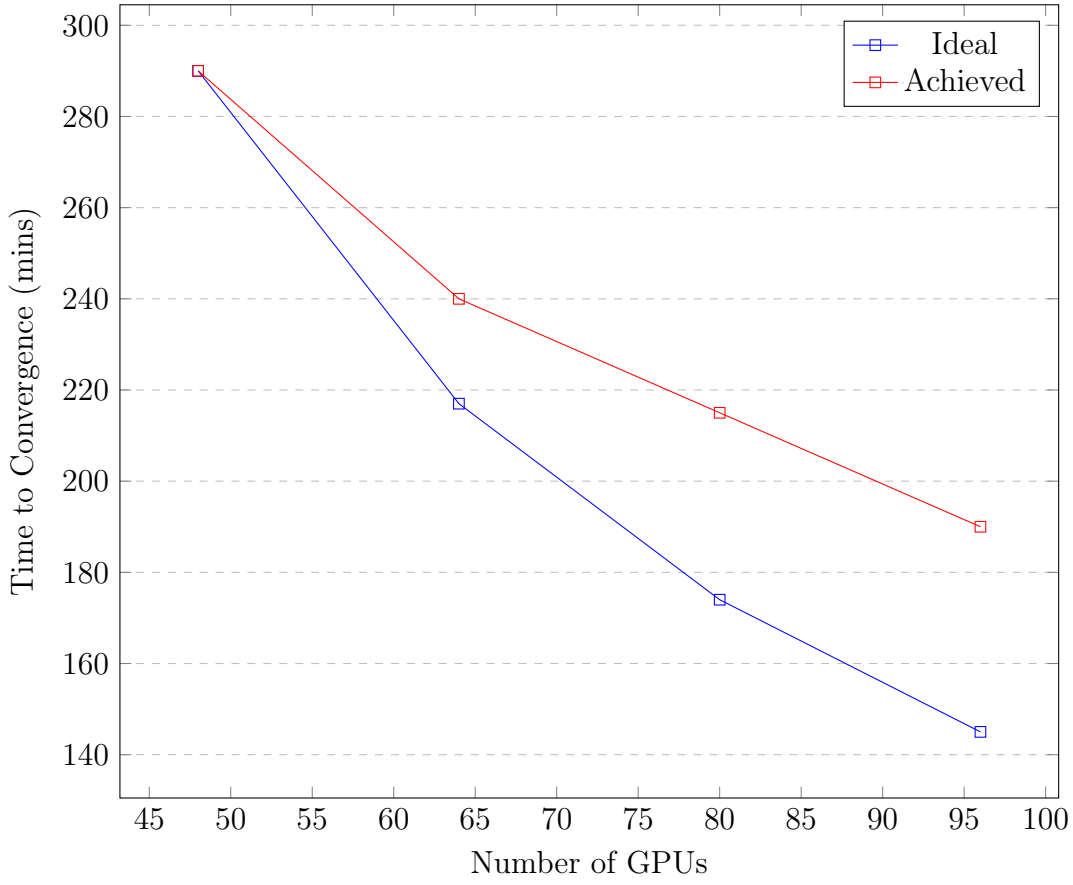

Figure 4-3: ResNet-50, ImageNet - Time to Convergence, local batch size=128

4.2 Impact of network load on performance

Often, distributed machine learning jobs run in cloud clusters and share the infrastructure with other jobs. This means the gradient update traffic is sharing the same links with other jobs and/or legacy data center traffic causing congestion that further induce stragglers. In this experiment, we test this assumption on Google Cloud Engine by creating controlled TCP flows across 2 of the 4 VMs in a distributed run. In this experiment, we measure throughput in images/sec/gpu of ResNet-50 benchmark running on 16 GPUs over 4 VMs. Figure 4-4 shows that by increasing the number of TCP flows across two of the VMs, one can significantly slow down the performance. With 30 TCP flows, the throughput drops to around 15% of the throughput in absence of any interfering TCP flows. This points to a greater need of creating distributed ML frameworks that are robust against crossing traffic in a shared cluster.

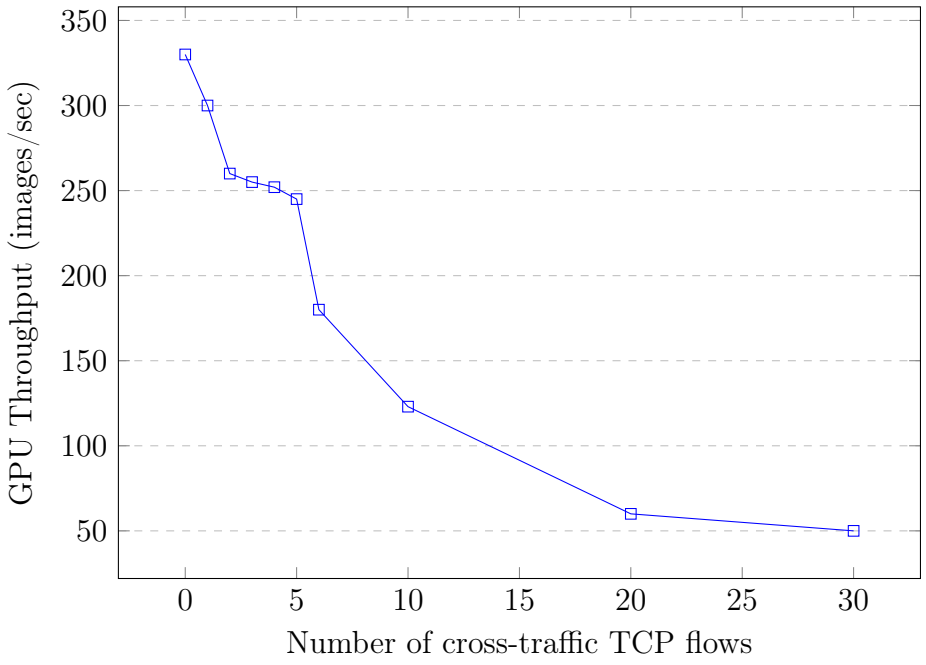

Figure 4-4: Impact of network congestion on performance, 16 GPUs, local batch size=128

4.3 Curbing the effect of stragglers using AIKIDO

In this set of experiments, we use AIKIDO to simulate straggler behavior on 1 of the 4 GPUs on a single Google Cloud VM running ResNet-50 on ImageNet [20, 14] and measure the Cumulative Distribution Function (CDF) of iteration duration times. Note that the ideal run here is a run without adding any simulated delay.

4.3.1 Straggler simulation with four GPUs

We use a delayConfig file to delay one pre-determined worker (worker ID=2) by 75ms every other iteration. Figure 4-5 shows that adding a 75 ms in this setup results in all iterations being somewhat delayed. While the overall shape of the CDF retains its step nature at Y=0.5, the other iterations are also delayed between 20-30ms. We believe that this is due to the interaction of compute and communication cycle in gradient updates. Delaying on one iteration can cause ripple effects into the next iteration being slow as well. More research is needed here to profile compute and communications cycle times and identify the non-linear interaction effects due to straggling workers. That being said, the aggregate behavior of straggler is consistent with the delayConfig that is being simulated here. More importantly, Figure 4-5 shows that both the *Skip* and *Active-Backup* strategies in AIKIDO are able to mitigate the straggler effect and bring the performance (iteration times) closer to the ideal.

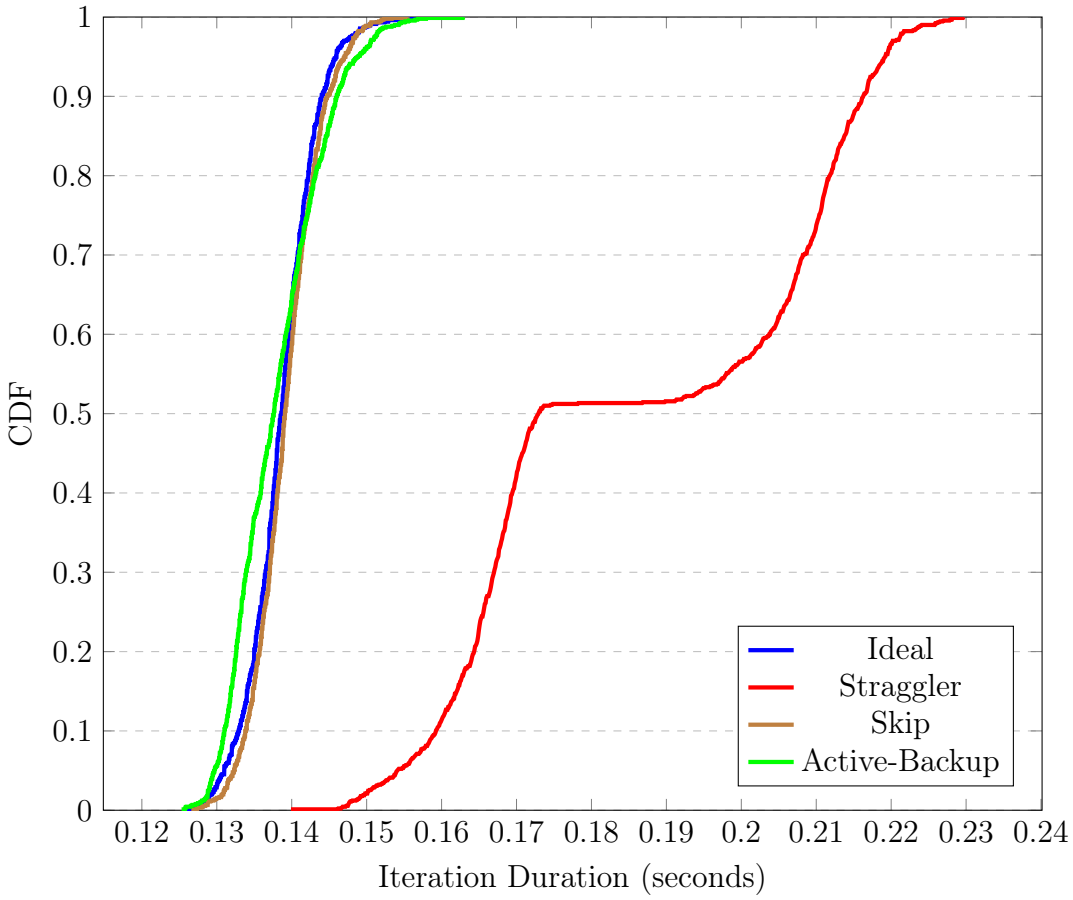

Figure 4-5: CDF of measured iteration duration for ResNet50 + ImageNet, 4 GPUs, global batch size=128

4.3.2 Straggler simulation with 16 GPUs

Similarly, we repeat the above experiment with the same delayConfig but instead scale the number of workers to 16 GPUs. The delayConfig here remains the same as that of previous experiment in which worker ID 2 is selected as the straggling node. Note that the ideal run here is a run without adding any simulated delay. For the Active-Backup experiment, we use a total of 16 GPUs in which AIKIDO designates 15 workers as primaries and 1 as an active backup. The primaries participate in the ring topology at every iteration, unless one of them straggles, in which case an active backup is swapped in its place. This allows AIKIDO to active provide gradient updates despite removing the straggler workers out of the ring.

In this run, the baseline iteration time is higher at around 175ms due to imperfect scaling in going from one VM with four GPUs to four VMs with 16 GPUs. We do not observe any spillover effect in this run. Again, we suggest that more future work is needed to understand the differences in compute and communications cycle as one scales to greater number of GPUs. Figure 4-6 shows both the *Skip* and *Active-Backup* strategies are able to mitigate the straggler and bring the performance close to the ideal with no stragglers.

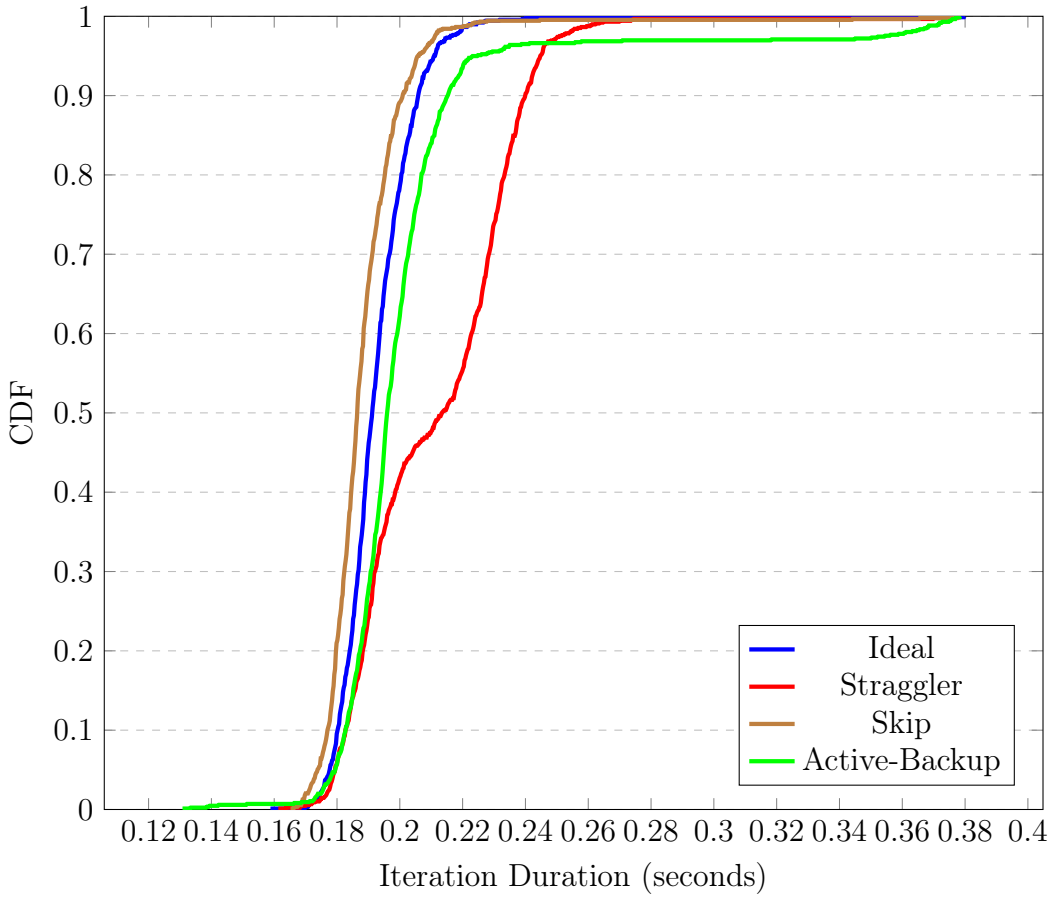

Figure 4-6: CDF of measured iteration duration for ResNet50 + ImageNet, 16 GPUs, global batch size 512.

4.3.3 Training ResNet-50 on ImageNet until convergence using simulated stragglers and AIKIDO

Finally, we test AIKIDO on a full run of ResNet-50 on ImageNet until convergence and measure how well *Skip* and *Active-Backup* strategies perform against the straggler and ideal runs. As demonstrated in Figures 4-7 and 4-8, both the loss functions and time to accuracy plot show that AIKIDO is able to curb the impact of stragglers while maintaining a low operational overhead. Table 4.3.3 summarizes our results from the full ResNet-50 + ImageNet run for various approaches.

Approach	Time to reach 92% top-5 accuracy
Ideal	392 mins
Straggler	444 mins
Skip	395 mins
Active	410 mins

Table 4.1: Summary of results on ResNet-50 + ImageNet run, local batch size=64, 16 GPUs

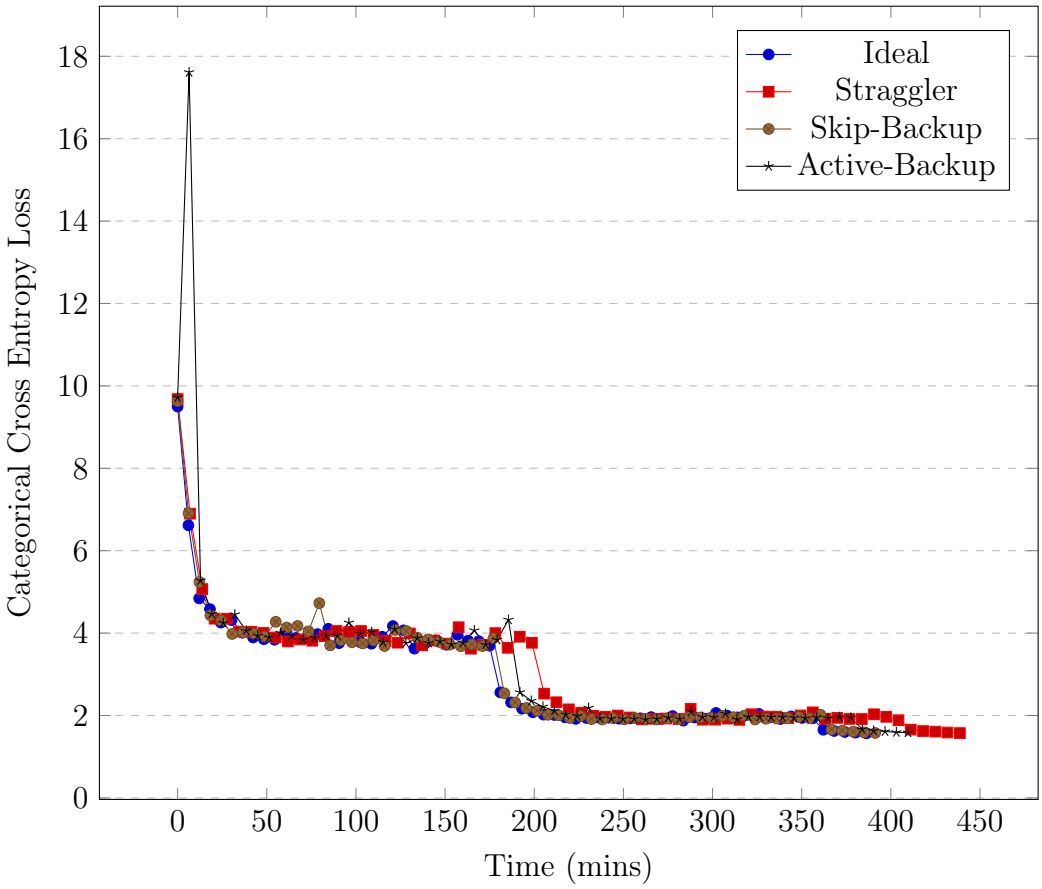

Figure 4-7: Categorical Cross Entropy Loss: ResNet50 + ImageNet on 16 V100 GPUs, global batch size 1024.

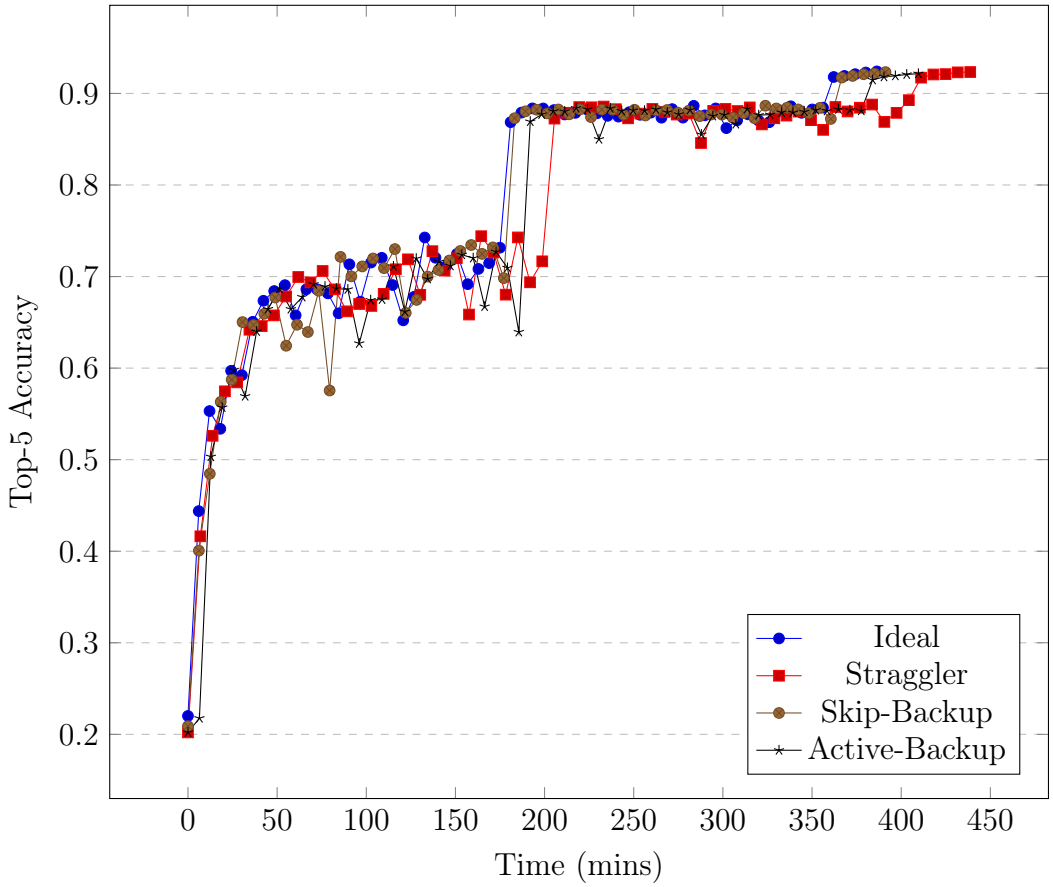

Figure 4-8: Top 5 Accuracy for ResNet50 training + ImageNet, 16 GPUs, global batch size 1024.

4.4 Chrome Profiler tool for visual inspection of the characteristics of training runs

We use AIKIDO profiler to analyse the amount of time that a training job is spent on communication time. We use four V100 GPU to train ResNet-50 on ImageNet. On this hardware, with hard-disk drive storage on Google VMs, we achieve average iteration times closer to 380ms. In this setting, we use the profiler to parse gradient wait times for a given chunk throughout the ResNet-50 training. Note that the waiting times are defined as the difference between the end of Reduce operation and the beginning of the broadcast operation (similar to Horovod's usage). Figure 4-9 shows the CDF of gradient wait times for 1000 iterations. Figure 4-9 shows that 50% of iterations spend over 80ms idle time waiting for gradients to arrive. We believe that this is largely due to software and scheduling inefficiencies. More research is needed to fully understand the involved bottlenecks to scaling distributed ML.

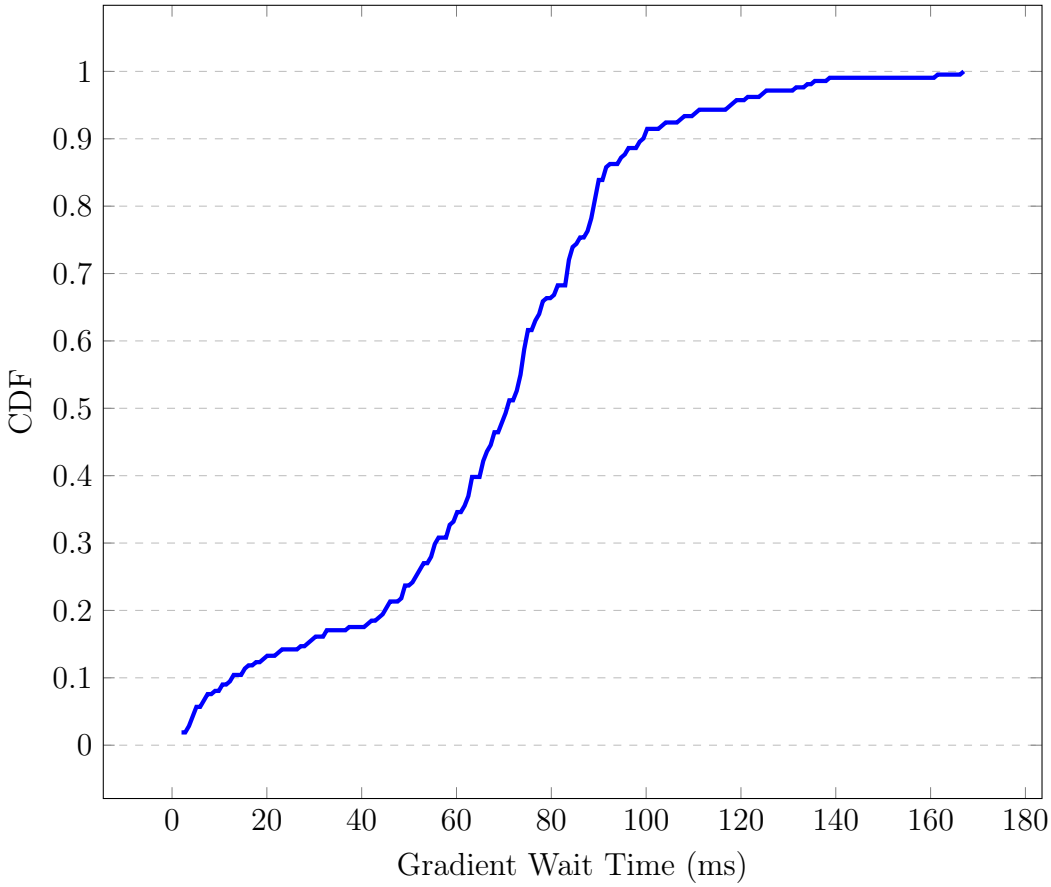

Figure 4-9: CDF of gradient wait times: ResNet-50, ImageNet, 4 GPUs, local batch size=128

4.5 Lessons Learned

Some of the key design choices for this project include using KungFu as our base distributed ML framework as opposed to Horovod. We faced a few significant challenges that we document here to aid future work.

- Non-Standard package for distributed ML: As a framework still in its experimental phase, KungFu has its set of quirks. For instance, KungFu rewrites its communications layer instead of using the standard MPI library. While their claim is that it helps in speeding up All-Reduce operation, it may be possible to work within the MPI framework and thus retain its reliability. For future work, we recommend the reader to avoid using KungFu and instead use Horovod as it has become the industry-standard for distributed ML. Furthermore, as with any standard framework, Horovod has more online support and documentation available than KungFu. We experienced several setbacks debugging a non-standard codebase.

- Tensorflow and Python versioning pitfalls: KungFu has known bugs and performance issues with certain Tensorflow version, and there is limited testing available for its performance beyond a few versions of TF and Python. Compatibility and performance guarantees are often demanded out of industry-level frameworrk, and therefore Horovod would be better suited for this purpose too. Additionally, we note that TensorFlow's new version - TF 2.0.0 - has eager execution mode on by default, and we faced compatibility and performance issues while working with it alongside KungFu.

- Google Cloud configuration pitfalls: In our Google Cloud Environment, we noticed a few issues in working with large scale distributed ML. One of the known issues in this project with KungFu was that of bottlenecks in scaling ResNet-50 training on ImageNet. We were able to fix it partially by switching

from Hard-disk drive to Solid-state drive for storing ImageNet, but still faced the issue at higher local batch sizes. While our suspicion is that it is an issue due to non-optimized TensorFlow data pipelining for loading ImageNet TFRecords into memory and pre-processing, further research is needed to completely profile these experiments and pin-point where the bottleneck lies.

Chapter 5

Conclusion

As the world embraces deep learning, training models are becoming ever more resource-intensive and distributed machine learning is gaining wider attention from researchers and professionals alike. In this thesis, we have shown that synchronous-SGD and AllReduce, while performing better than their counterparts, still present a unique set of challenges at scale. As we scale machine learning jobs in data centers to hundreds of workers, the problem of stragglers needs to be solved. AIKIDO provides a first step towards curbing the impact of network-induced stragglers on distributed machine learning training jobs. More research is required to fully protect against stragglers in modern heterogeneous data center environments.

Bibliography

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

[2] Takuya Akiba, Shuji Suzuki, and Keisuke Fukuda. Extremely large minibatch sgd: Training resnet-50 on imagenet in 15 minutes. `https://arxiv.org/abs/1711.04325`, 2017.

[3] Julaiti Alafate and Yoav Freund. Tell me something new: A new framework for asynchronous parallel learning. `https://arxiv.org/abs/1805.07483`, 2018.

[4] 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.

[5] Robin Andersson, Jim Dowling, Ermias Gebremeskel, and Kim Hammar. Goodbye horovod, hello collectiveallreduce, Oct 2018. `https://www.logicalclocks.com/blog/goodbye-horovod-hello-collectiveallreduce`.

[6] Charles J Archer, Gabor Dozsa, Joseph D Ratterman, and Brian E Smith. Performing an allreduce operation using shared memory, April 17 2012. US Patent 8,161,480.

[7] Ali Bakhoda, George L Yuan, Wilson WL Fung, Henry Wong, and Tor M Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.

[8] Léon Bottou and Olivier Bousquet. The tradeoffs of large scale learning. In *Advances in neural information processing systems*, pages 161–168, 2008.

[9] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. `https://arxiv.org/abs/1604.00981`, 2016.

[10] Trishul 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)*, pages 571–582, 2014.

[11] Guoxin Cui, Jiafeng Guo, Yixing Fan, Yanyan Lan, and Xueqi Cheng. Trend-smooth: Accelerate asynchronous sgd by smoothing parameters using parameter trends. *IEEE Access*, 7:156848–156859, 2019.

[12] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. High-performance distributed ml at scale through parameter server consistency models. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[14] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[15] Farshid Farhat, Diman Zad Tootaghaj, Yuxiong He, Anand Sivasubramaniam, Mahmut Kandemir, and Chita R. Das. Stochastic modeling and optimization of stragglers. *IEEE Transactions on Cloud Computing*, 6(4):1164–1177, oct 2018.

[16] Message Passing Forum. mpi: A message-passing interface standard. Technical report.

[17] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch, and domain parallelism in training neural networks. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures - SPAA '18*. ACM Press, 2018.

[18] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[19] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 98–111, New York, NY, USA, 2016. ACM.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[21] Qirong Ho, James Cipar, Henggang Cui, Seunghak Lee, Jin Kyu Kim, Phillip B Gibbons, Garth A Gibson, Greg Ganger, and Eric P Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *Advances in neural information processing systems*, pages 1223–1231, 2013.

[22] Yuzhen Huang, Tatiana Jin, Yidi Wu, Zhenkun Cai, Xiao Yan, Fan Yang, Jinfeng Li, Yuying Guo, and James Cheng. Flexps: Flexible parallelism control in parameter server architecture. *Proceedings of the VLDB Endowment*, 11(5):566–579, 2018.

[23] Forrest N. Iandola, Khalid Ashraf, Matthew W. Moskewicz, and Kurt Keutzer. Firecaffe: near-linear acceleration of deep neural network training on compute clusters. *CoRR*, abs/1511.00175, 2015.

[24] Manya Ghobadi Muriel Médard James Salamy, Ayush Sharma. Flexent: Entropy coding to curb stragglers in large-scale distributed machine learning. In *SOSP AI Systems Workshop*, 2019.

[25] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *CoRR*, abs/1807.11205, 2018.

[26] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th Symposium on Operating Systems Principles*, SOSP '19, 2019.

[27] Kaggle. `https://www.kaggle.com/c/dogs-vs-cats`.

[28] Can Karakus, Yifan Sun, Suhas Diggavi, and Wotao Yin. Straggler mitigation in distributed optimization through data encoding. In *Advances in Neural Information Processing Systems*, pages 5434–5442, 2017.

[29] Idit Keidar. ACM SIGACT news distributed computing column 32: the year in review. *ACM SIGACT News*, 39(4):53–54, 2008.

[30] Kangwook Lee, Maximilian Lam, Ramtin Pedarsani, Dimitris Papailiopoulos, and Kannan Ramchandran. Speeding up distributed machine learning using codes. *IEEE Transactions on Information Theory, NeurIPS Workshop on Machine Learning Systems, and IEEE International Symposium on Information Theory (ISIT)*, 2015.

[31] 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.

[32] Zongpeng Li and Baochun Li. Efficient and distributed computation of maximum multicast rates. In *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3, pages 1618–1628. IEEE, 2005.

[33] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[34] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 5330–5340, 2017.

[35] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. `https://arxiv.org/abs/1710.06952`, 2017.

[36] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. Deep learning with dynamic computation graphs. `https://arxiv.org/abs/1702.02181`, 2017.

[37] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[38] Andrei-Octavian Brabete Alexandros Koliousis Peter Pietzuch Luo Mai, Guo Li. Adaptive distributed training of deep learning models. In *SOSP AI Systems Workshop*, 2019. 2-Page Abstract: `http://learningsys.org/sosp19/assets/papers/13_CameraReadySubmission_camera_ready.pdf` and Base commit: `https://github.com/lsds/KungFu/commit/6f22b4f457bb5ebebd9400a9a3eda673228d6db8`.

[39] Hiroaki Mikami, Hisahiro Suganuma, Pongsakorn U.-Chupala, Yoshiki Tanaka, and Yuichi Kageyama. Imagenet/resnet-50 training in 224 seconds. *CoRR*, abs/1811.05233, 2018.

[40] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association*, 2010.

[41] David Kung Hillery Hunter Minsik Cho, Ulrich Finkler. Blueconnect: Decomposing all-reduce for deep learning on heterogeneous network hierarchy. SysML Conference, 2019.

[42] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q.

Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[43] Shar Narasimhan. Sr. product manager, ai, nvidia. `https://devblogs.nvidia.com/training-bert-with-gpus/`, August 2019.

[44] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[45] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in scala*. Artima Inc, 2008.

[46] 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.

[47] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[48] Netanel Raviv, Itzhak Tamo, Rashish Tandon, and Alexandros G. Dimakis. Gradient coding from cyclic mds codes and expander graphs. *International Conference on Machine Learning*, 2018.

[49] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In J. Shawe-Taylor, R. S. Zemel, P. L. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 693–701. Curran Associates, Inc., 2011.

[50] Sebastian Ruder. An overview of gradient descent optimization algorithms. `https://arxiv.org/abs/1609.04747`, 2016.

[51] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. Scaling distributed machine learning with in-network aggregation.

[52] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. `https://arxiv.org/pdf/1802.05799.pdf`, 2018.

[53] Hang Shi, Yue Zhao, Bofeng Zhang, Kenji Yoshigoe, and Athanasios V. Vasilakos. A free stale synchronous parallel strategy for distributed machine learning. In *Proceedings of the 2019 International Conference on Big Data Engineering (BDE 2019) - BDE 2019*. ACM Press, 2019.

[54] Chromium Open Source. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.

[55] TensorFlow Open Source. https://github.com/tensorflow/models/tree/master/official/vision/image_classification/.

[56] Rashish Tandon, Qi Lei, Alexandros G. Dimakis, and Nikos Karampatziakis. Gradient coding: Avoiding stragglers in distributed learning. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3368–3376, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. http://proceedings.mlr.press/v70/tandon17a.html.

[57] Maarten Van Steen and A Tanenbaum. Distributed systems principles and paradigms. *Network*, 2:28, 2002.

[58] Dean Wampler and Alex Payne. *Programming Scala: Scalability= Functional Programming+ Objects*. " O'Reilly Media, Inc.", 2014.

[59] Da Wang, Gauri Joshi, and Gregory W. Wornell. Efficient straggler replication in large-scale parallel computing. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 4(2):7:1–7:23, April 2019.

[60] Matthias Wiesmann, Fernando Pedone, André Schiper, Bettina Kemme, and Gustavo Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474. IEEE, 2000.

[61] Yang You, Jing Li, Jonathan Hseu, Xiaodan Song, James Demmel, and Cho-Jui Hsieh. Reducing bert pre-training time from 3 days to 76 minutes. *arXiv preprint arXiv:1904.00962*, 2019.

[62] Hsiang-Fu Yu, Cho-Jui Hsieh, and Inderjit S. Dhillon. Parallel asynchronous stochastic coordinate descent with auxiliary variables. In Kamalika Chaudhuri and Masashi Sugiyama, editors, *Proceedings of Machine Learning Research*, volume 89 of *Proceedings of Machine Learning Research*, pages 2641–2649. PMLR, 16–18 Apr 2019.

[63] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association.

[64] Shen-Yi Zhao and Wu-Jun Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[65] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhiming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation for distributed deep learning. *arXiv preprint arXiv:1609.08326*, 2016.

[66] 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.