UNIVERSITY OF CALIFORNIA,
IRVINE

Hierarchical Heterogeneous Cluster Systems for Scalable Distributed Deep Learning

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Yibo Wang

Dissertation Committee:
Professor Jean-Luc Gaudiot, Chair
Professor Nader Bagherzadeh
Professor Sergio Gago-Masague

2023

DEDICATION

Dedicated to my parents, whose unwavering support has been a guiding force in my academic journey. To my loving wife and kids, you are my constant inspiration and the joy that brightens every step of the way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to thank my supervisor and committee chair, Professor Jean-Luc Gaudiot for his invaluable supervision, and continuous support during the course of my PhD degree. I would also like to thank my committee members Professor Nader Bagherzadeh and Professor Sergio Gago-Masague for their valuable advice and suggestions on my research work.

Additionally, I would also like to thank Professor G. P. Li and Dr. Yutian Ren from Calit2 - UCI, for providing the resources and valuable comments for my research experiments.

Furthermore, I would like to thank my colleagues in the PArallel Systems Computer Architecture LAB (PASCAL), Dr. Tongsheng Geng, Dr. Congmiao Li, Dr. Beverly Abadines Quon, Dr. Nazanin Ghasemian Moghaddam, and Jaya Keshava Chandra Kotha, for the assistance and support of my research work and cherished time spent together.

Finally, many thanks to faculty and staff from Graduate Division, Department of Electrical Engineering and Computer Science for years of support of my research and study.

This work was partially supported by the National Science Foundation under award CCF-176379.

VITA

Yibo Wang

EDUCATION

Doctor of Philosophy in Electrical and Computer Engineering **2023**
University of California, Irvine *Irvine, California*

Master of Science in Computer Science **2017**
University of California, Irvine *Irvine, California*

RESEARCH EXPERIENCE

Graduate Research Assistant **2019–2023**
University of California, Irvine *Irvine, California*

Research Specialist **2017–2019**
Calit2-UCI *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2019–2023**
University of California Irvine *Irvine, CA*

ABSTRACT OF THE DISSERTATION

Hierarchical Heterogeneous Cluster Systems for Scalable Distributed Deep Learning

By

Yibo Wang

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2023

Professor Jean-Luc Gaudiot, Chair

Distributed deep learning framework should aim at high efficiency of training and inference of distributed exascale deep learning algorithms. There are three major challenges in this endeavor: scalability, adaptivity and efficiency. Any future framework will need to be adaptively utilized for a variety of heterogeneous hardware and network environments and will thus be required to be capable of scaling from single compute node up to large clusters. Further, it should be efficiently integrated into popular frameworks such as TensorFlow, PyTorch, etc.

This dissertation proposes a dynamically hybrid (hierarchy) distribution structure for distributed deep learning, taking advantage of flexible synchronization on both centralized and decentralized architectures, implementing multi-level fine-grain parallelism on distributed platforms. It is scalable as the number of compute nodes increases, and can also adapt to various compute abilities, memory structures and communication costs.

Chapter 1

Introduction

The introduction section will provide an overview of the current situation in machine learning, particularly focusing on the scale of training and the trend toward distributed systems. It will cover the fundamentals of neural networks, with a specific emphasis on fully connected layers and the reasons why we need to consider adaptivity, scalability and performance issues and why they are crucial during the training process.

Neural networks have triggered a remarkable transformation in various aspects of our daily lives. They have found applications in a wide array of fields, such as natural language processing (NLP) [12], optics [17], image processing [30], and computer vision (CV) [4]. These developments have driven progress in technologies like autonomous driving [6], face recognition [34], anomaly detection [24], text comprehension [22], and even art [13]. The influence of neural networks continues to expand and strengthen.

At present, every research domain has been touched by this wave of innovation in neural networks, resulting in substantial improvements in capabilities and performance. The primary drivers behind their current success are the vast volumes of available data, which are crucial for training large neural networks, and the advancements in GPU computing, which

dramatically accelerate training times (sometimes achieving up to a 100-fold speed improvement compared to traditional CPUs). The advantages of neural networks are particularly pronounced at a large scale. Therefore, having a large data size and the necessary hardware for processing them are essential for the success.

1.1 Current Trends of DNNs in Distributed Systems

Realistic quantities of training data range from 1TB to 1PB. This allows one to create powerful and complex models with 10^9 to 10^{12} parameters. [19] These models are often shared globally by all worker nodes, which must frequently access the shared parameters as they perform computation to refine it. As shown in Figure 1.1, we can observe a significant increase starting from mid 2018. The growth rate before 2018 was 0.1OOMs/year(order of magnitudes per year), whereas after 2018 the growth rate is 0.9OOMs/year if we assume a single exponential trend. [32]

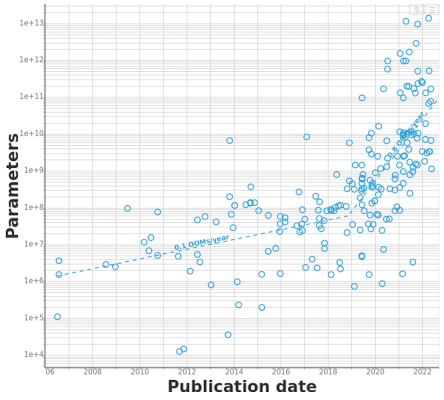
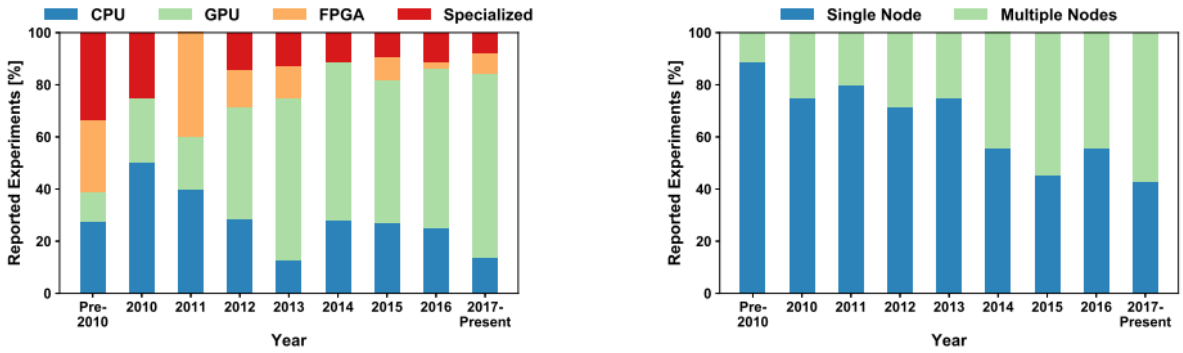


Figure 1.1: Trends in Model Size [32]

In the amount of training data, the model accuracy can, to a large extent, be improved by feeding more training data into the model In practice, it is reported that 10s to 100s of Terabyte (TB) of training data are used in the training of a neural network model. [19] In the size and complexity of the models themselves, simple and shallow neural network has

evolved to increasing depth and more sophisticated model architecture to improve the model accuracy. For example, as early as 2010, Ciresan et al. [8] has been proved that increasing the hidden layers and many neurons per layer improves the accuracy rate on the MNIST data set. In the aspect of infrastructure, the availability of programmable highly-parallel hardware, especially graphics processing units (GPUs), is a key-enabler to training large models with a lot of training data within a limited time.

Figure 1.2 shows a concise summary of the machine architectures featured in research papers over the years. Not only the GPUs prevalence in the recent year, it is important to note that even accelerated nodes may not fully meet the demands of the extensive computational workloads. Figure 1.2 (b) visually illustrates the rapid growth of multi-node parallelism in these research endeavors, signaling that, commencing in 2015, distributed-memory architectures equipped with accelerators like GPUs have become the default choice for machine learning across all scales in the present day.

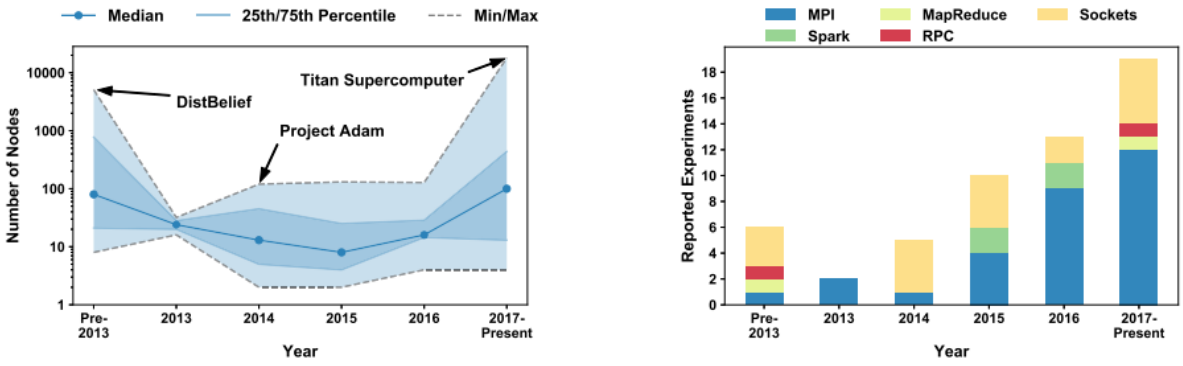


(a) Hardware Architectures (b) Training with Single vs. Multiple Nodes

Figure 1.2: Parallel Architectures in Deep Learning. [3]

Figure 1.3 shows how the number of nodes used in deep learning research has evolved over time. There was initially a large number of nodes during the DistBelief era, a slight dip when powerful accelerators were introduced, and a steady increase since 2015, mainly due to large-scale deep learning. It's important to note that configurations resembling those used in High-Performance Computing (HPC) have become common in modern training scenarios.

In the same figure, the various communication methods used in 55 out of the 80 papers that employ multi-node parallelism can be observed. This indicates that the research community quickly has realized the similarities between deep learning and large-scale HPC applications. As a result, starting in 2016, the established MPI interface became the standard for communication in distributed deep learning.



(a) Node Count (b) Communication Layer

Figure 1.3: Parallel Nodes and Communication Layer in Deep Learning. [3]

The primary benefit of fully connected networks is that they are "structure-agnostic" [16] in nature, which means specific assumptions are not required about the input data. While this characteristic makes fully connected networks versatile and widely applicable, these networks often exhibit lower performance compared to specialized networks designed to match the inherent structure of a given problem domain. In many applications, multi-feature and multi-layer perceptron neural networks are preferred because the features are known in advance during data collection. Feature engineering can be conducted prior to the use of the fully connected layer, which is computationally intensive in these scenarios. In contrast to Convolutional Neural Networks (CNNs) which have found extensive applications in areas such as image and video processing, there are situations where the convolution layer may not be suitable. For instance, in certain medical contexts, like the study by Haoren Wang et al. [33] arrhythmia classification was achieved by manually extracting features and employing fully connected layers.

1.2 Problem Statement, Motivation and Goals

The main challenge we encounter is the use of computational power and network resources in distributed systems for scalable machine learning and deep learning to tackle parallel computing problems. When we specifically focus on fully connected layers in neural networks, where both data and models are exascale, it is crucial to coordinate computation, communication, and data storage to ensure the efficient utilization of all resources. To achieve this, we aim to establish a hierarchical approach for displaying and managing these resources.

We are particularly interested in the inclusion of heterogeneity into the distributed system considerations because it is inevitable when we aim to utilize various hardware resources, such as different generations of Intel CPUs, ARM CPUs, and different brand or series of AMD or Nvidia GPUs. We also explore the integration of CPUs into GPU training, even though GPUs are prevalent in the training process. We seek ways for CPUs to contribute to the process, if possible. It is also worth considering the practicality of multiple clusters or data centers collaborating, and when dealing with heterogeneous hardware, establishing proper connections becomes essential.

Since the model is distributed across different nodes, we are actively researching the design of a distributed architecture that can efficiently coordinate and enhance resource utilization. Additionally, the choice of an appropriate synchronization method plays a crucial role in overall training performance. This is because local model parameters need to be computed, loaded, or saved in local storage, and these actions must be synchronized effectively.

Chapter 2

Background

Extensive research has been conducted in this rapidly advancing field, with contributions from various research communities. This is due to the numerous challenges associated with managing large, distributed infrastructures for neural networks and hosting a vast number of models trained with substantial training data. Many hot research areas involve exploring questions related to parallelization, resource scheduling, elasticity, data management and portability, distributed and networked systems, and more.

In this chapter, we will delve into several critical aspects, including the fundamentals of neural network training (both forward and backward propagation), distributed infrastructures, parallelization in neural network training, synchronization, fine-grain model parallelism, and various platforms and optimizations.

2.1 Forward and Backward Propagation

Forward propagation, also known as the forward pass, calculates and stores intermediate variables, moving from the input to the output layer in a neural network. Backward Propa-

gation, on the other hand, is the preferred method for adjusting the weights to minimize the loss function. In this section, we'll examine Forward and Backward Propagation in detail, understanding their mathematical principles and why Backward Propagation is the preferred approach.

2.1.1 Forward Propagation Calculus

For simplicity, we choose to use a three layer architecture of the neural network is [2, 3, 1] with: 2 independent variables, X_i in the input layer, 3 nodes in the hidden layer, and 1 node in the output layer. The neural network operates by:

a. Initializing the weights with some random values, which are mostly between 0 and 1.

b. Compute the output to calculate the loss or the error term.

c. Adjust the weights so that to minimize the loss.

We repeat these three steps until have reached the optimum solution of the minimum loss function or exhausted the pre-defined epochs (i.e. the number of iterations).

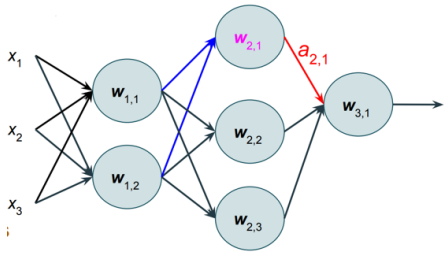


Figure 2.1: A Three Layers Neural Network

Now, the computation graph after applying a nonlinear activation function is:

$$a_{2,1} = \sigma(a_1 \cdot w_{2,1}) \tag{2.1}$$

σ is the activation function, e.g. Sigmoid function show in Figure 2.2

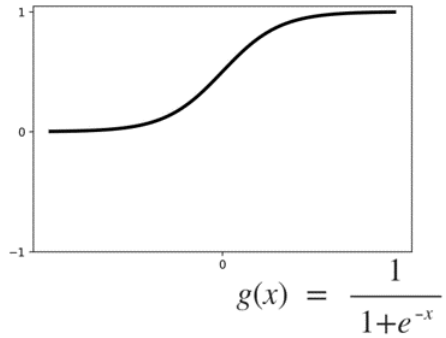


Figure 2.2: Sigmoid Function

The output of this three layers of neural network will be:

$$ouput = \sigma(\sigma(\sigma(X \cdot W_1) \cdot W_2) \cdot W_3) \tag{2.2}$$

Various activation functions are available, and there's no definitive guide to choosing the best one for a specific problem. It typically involves a trial-and-error approach, where experiment with different functions to see which one performs best for particular problem. Here are four of the most commonly used activation functions:

a. Sigmoid function (σ): $g(z) = 1/(1 + e^{-z})$. The sigmoid function is best for the output layer as it keeps values between 0 and 1 for easy interpretation as probabilities. However, using it in hidden layers can slow down learning because it has a very small gradient over a large part of its input range.

b. Hyperbolic Tangent function: $g(z) = (e^z - e^{-z})/(e^z + e^{-z})$. The tanh function is an improvement over the sigmoid because it centers the output around zero, helping with faster learning. However, like the sigmoid, it still faces the issue of having a very small gradient over a significant part of its input range.

c. Rectified Linear Unit (ReLU): $g(z) = \max\{0, z\}$. ReLU is a good choice for many problems because it's similar to linear functions, which are easy to optimize. However, it

has a limitation: the gradient is zero for values less than or equal to zero, making learning difficult in those cases.

d. Leaky Rectified Linear Unit: $g(z) = \max\{\alpha z, z\}$. It overcomes the zero gradient issue from ReLU and assigns α which is a small value for $z \leq 0$.

ReLU is a commonly used choice for hidden layers. For binary classification in the output layer, the sigmoid function is a natural and fitting option.

2.1.2 Backward Propagation Calculus

Backward Propagation is an algorithm for calculating the gradient of the cost function of a network.

$$\nabla C = \begin{cases} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{cases} \tag{2.3}$$

In backpropagation, we are primarily concerned with the derivatives of the cost function concerning the various weights and biases in the network. These derivatives are the components of the gradient vector, and they guide us in optimizing the network. For Simplicity, we use three layers of one neuron network as an example:

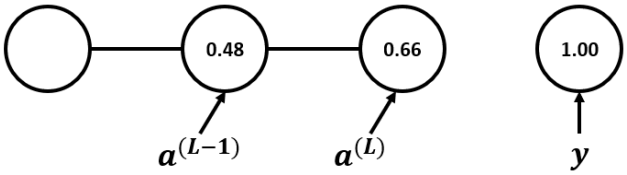


Figure 2.3: A Three Layers of Single Neuron Neural Network

$$\frac{\partial C}{\partial w^L} = \frac{\partial z^L}{\partial w^L}\frac{\partial a^L}{\partial z^L}\frac{\partial C}{\partial a^L} \tag{2.4}$$

where

$$\frac{\partial C}{\partial a^L} = 2(a^L - y), \qquad \frac{\partial a^L}{\partial z^L} = \sigma'(z^L), \qquad and \qquad \frac{\partial z^L}{\partial w^L} = a^{L-1} \tag{2.5}$$

Update the all the weights

$$W = W + \alpha \nabla C \tag{2.6}$$

where α is learning rate.

In general for a L layers neural network , the gradient will be calculated as:

$$\nabla C = \frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \sigma'(z^l_j)\frac{\partial C}{\partial a^l_j} \tag{2.7}$$

where

$$\frac{\partial C}{\partial a^l_j} = \Sigma_{j=0}^{n_{l+1}-1} w^{(l+1)}_{jk} \sigma'(z^{(l+1)}_j)\frac{\partial C}{\partial a^{(l+1)}_j} \tag{2.8}$$

In the context of distributed training, this gradient will be generated from each worker, and shared among the workers, and then weights can be updated using the up-to-date gradient on each workers. We will get into the details of the parallel methods that used in the distributed neural network training.

2.2 Parallelization Methods

In deep neural network training, there are three dominant parallelization method: data parallelism, model parallelism, pipeline parallelism, and also hybrid forms of parallelism.

2.2.1 Data Parallelism

In data parallelism, multiple workers (like machines or GPUs) each use a copy of the same deep neural network model (as shown in 2.4). The training data is divided into separate chunks and given to the model copies on the workers for training. Each worker trains on its data chunk, which leads to updates in the model parameters. So, the model parameters among the workers must be kept in synchronization. Data parallelism offers a significant benefit in that it can be used with any deep learning model architecture without any specific knowledge about the model. It's highly effective for tasks that demand substantial computational power but involve only a limited number of parameters, like Convolutional Neural Networks (CNNs). However, performance bottleneck occurs when dealing with operations that have a large number of parameters, as the synchronization of these parameters becomes a limiting factor. [21] The issue can be addressed by using larger batch sizes, but this can introduce data delay among the workers and result in poor model convergence. Another drawback of data parallelism is that it cannot handle the case when the model is too large to fit on a single device. Additionally, in many data parallel training methods, it is necessary that the training data is independent and identically distributed (i.i.d.) [37], so it allows the updates from parallel workers can be easily combined to update the global model parameters.

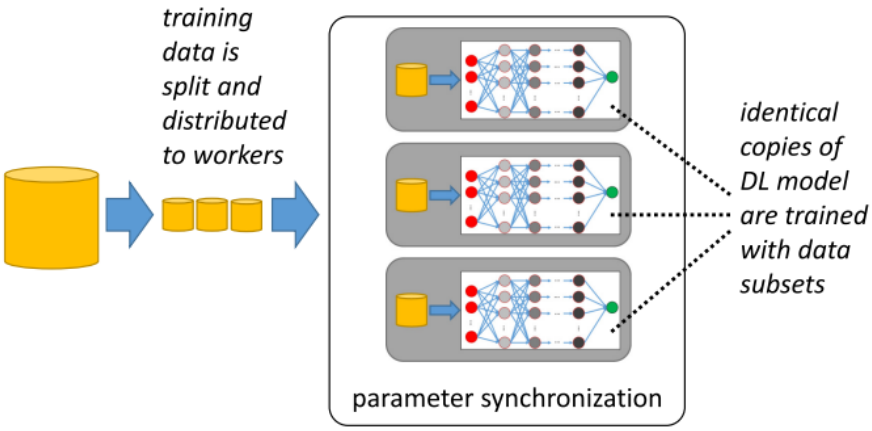


Figure 2.4: Data Parallelism [26]

2.2.2 Model Parallelism

In model parallelism, the deep neural network model is divided, and each worker focuses on training a specific part of the model, as shown in Figure 2.5. Workers handling the input layer get the training data. During the forward pass, the output is calculated and then sent to workers managing the next layer. In the backpropagation phase, gradients are computed, starting at the workers responsible for the output layer and moving back towards the workers handling the input layers.

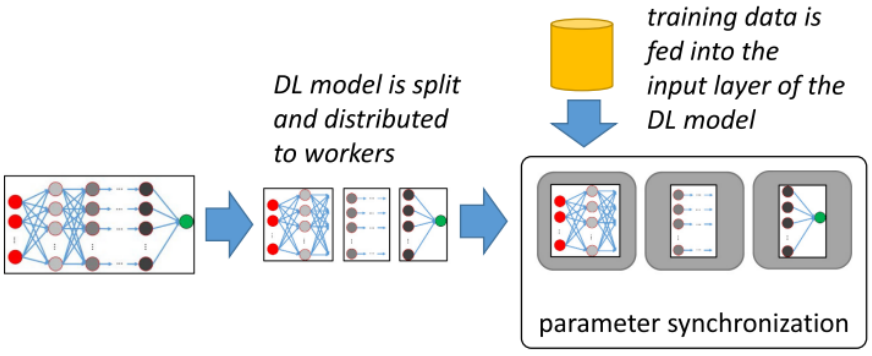


Figure 2.5: Model Parallelism [26]

The biggest challenge in model parallelism is figuring out how to split the model among the parallel workers. One common way to find the best model splitting is by using reinforcement learning. [28] It starts with an initial partition, tries different permutations, and keeps the

ones that improve performance, repeating this process until performance stabilizes. The key benefit of model parallelism is its minimal memory requirements. Since the model is divided, each worker needs less memory. This is particularly valuable when the entire model is too large to fit on a single device, which can happen when using specialized hardware like GPUs or TPUs. The disadvantages of model parallelism are the heavy communication between workers. Splitting complex models effectively can be challenging, leading to potential worker slowdowns due to communication and synchronization issues. As a result, increasing the complexity of model parallelism doesn't always speed up training. [28]

2.2.3 Hybrid Parallelism

Hybrid Parallelism is widely used in deep learning because complex models often have diverse layers with different architectures, requiring various parallelization methods. As a result, hybrid approaches that combine data, model, and pipeline parallelism are commonly employed to efficiently train these models. For instance, Krizhevsky [23] suggested using data parallelism for computationally intensive layers like convolutional and pooling layers with few parameters, and model parallelism for fully connected layers, which are lighter in computation but have many parameters. Additionally, instead of relying solely on manually designed parallelization optimizations, recent developments include automated optimization approaches. One such example is FlexFlow by Jia et al. [21], which is an automatic parallelization optimizer using an execution simulator. It accurately predicts the performance of parallelization strategies and is faster than previous methods that executed each strategy one by one.

2.2.4 Pipeline Parallelism

Pipeline parallelism is one example of hybrid parallelism, which utilizes both model parallelism and data parallelism. In pipeline parallelism, the model is divided among workers and each worker handles a different part of the model (as shown in Figure 2.6). The training data is also split into batches. Each worker processes batches, and passing results to the subsequent workers. In this way, multiple batches are streamed through the forward and backward passes in parallel, increasing worker efficiency compared to pure model parallelism, where only one batch is processed at a time. At the same time, This approach retains the advantages of model parallelism, where no single worker needs the entire model. Techniques like GPipe [20] and PipeDream [18] support pipeline parallelism.

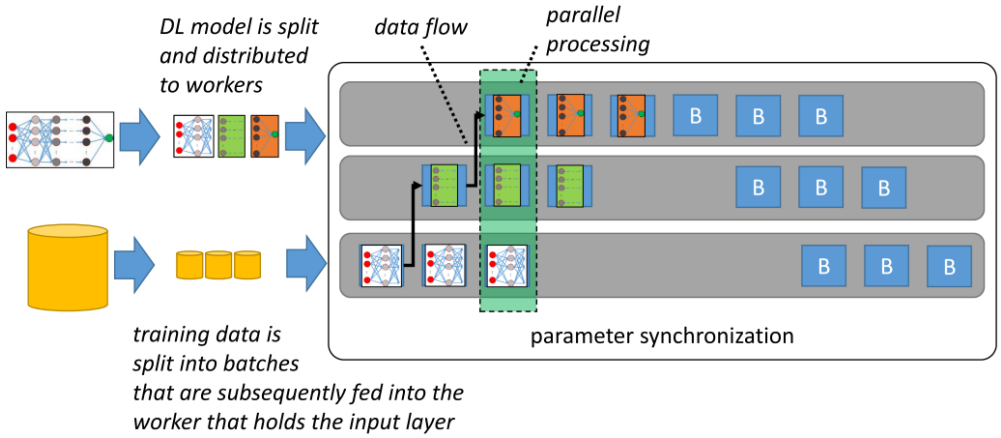


Figure 2.6: Pipeline Parallelism [20]

2.3 Parameter Distribution and Communication

There are three key challenges of creating an effective system architecture for parameter distribution and communication. First, scalability ensures the architecture can handle a large number of parallel workers that update the deep learning model and receive updates for further training. Second, configuration simplicity makes it easy to set up and config-

ure the system to achieve good performance without requiring extensive parameter tuning. Third, optimal use of primitives makes efficiently using lower-level tools like communication primitives, such as NCCL, to enhance performance.

2.3.1 Centralized

The centralized parameter server is the most popular architecture in data parallel neural network. The workers in the system are logically connected to a central parameter server, and periodically report their computed parameters or parameter updates to a set of parameter servers (as shown in Figure 2.7). Systems that use parameter server architecture includes MapReduce [11], TensorFlow [1], SparkNet [29], Poseidon [40], GeePS [9], DistBelief [10], and etc. Sharding of the model parameters and distribute the shards on multiple parameter servers [10], is a common approach is to update in parallel.

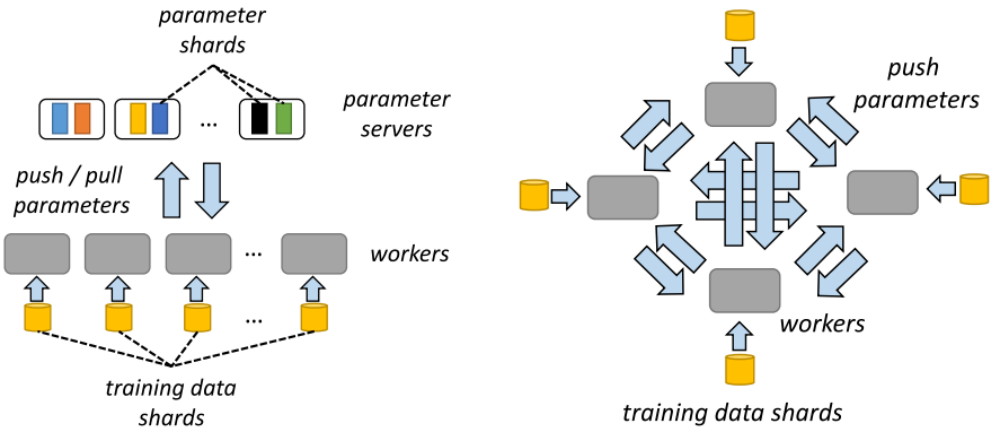


Figure 2.7: Centralized and Decentralized Architecture

2.3.2 Decentralized

In a decentralized architecture, there's no parameter server. Workers directly exchange parameter updates through an allreduce operation, as shown in Figure 2.7. The way workers

are connected matters. In a fully connected network where every worker talks to every other worker, the communication cost scales with the square of the worker count ($O(n^2)$ with n workers), creating a communication bottleneck.

A common alternative is to employ a ring topology (referred to chapter 3 for detailed discussion about ring-allreduce) which reduce the complexity of the communication cost to be independent of the worker count. Other topologies that have been proposed such as, a tree topology [2], and a graph that is built based on a Halton sequence [25], dynamically changed run-time topologies [35], and etc.The primary drawback of topologies other than the fully connected one is that they can require more time for parameter updates to propagate to all workers because there may be multiple hops between worker pairs. However, adjusting the worker topology isn't the only method to reduce network load. Watcharapichat [36] use a fully connected network of workers, but break down the gradients exchanged between workers into partitions (partial gradient exchange). The communication load depends on the partition size, determined by the number of partitions, and the number of workers. To keep network bandwidth constant, the number of partitions adjusts automatically, regardless of how many workers are involved.

2.3.3 Comparison of Centralized and Decentralized

In this section we summarize the advantages and disadvantages for the comparison of centralized and decentralized architecture.

The advantages of centralized are summarized as following:

1. Easy control and cost efficient. Centralized networks make reporting, securing, monitoring, and management simple.

2. Elastic scalability. New nodes can be added without restarting the running framework.

The disadvantage of centralized are summarized as following:

1. Inconveniences of implementing and tuning a parameter server.

2. Potential bottleneck when the network traffic is large.

3. Lost compute power when assign the nodes are used for parameter server.

The advantages of Decentralized are summarized as following:

1. Avoids the inconveniences of implementing and tuning a parameter server.

2. Fault tolerance can be achieved more easily, because there is no single point of failure such as the parameter server.

The disadvantage of Decentralized are summarized as following:

1. Communication increases quadratically with the number of workers, if there is no countermeasures are taken.

2. The lack of vertical top-down visibility, and oversight and analytics throughout the network can make scaling up or global changes challenging.

In summary, both centralized and decentralized approaches are commonly integrated into open-source deep learning frameworks. Some frameworks, like TensorFlow and MXNet, support both approaches. For example, TensorFlow utilizes the decentralized architecture for training on a single compute node with multiple GPUs, taking advantage of efficient allreduce implementations like NCCL allreduce. However, when it comes to multi-node training, the centralized architecture is often employed.

2.4 Synchronization

Determining when to synchronize parameters among parallel workers has been a subject of significant focus. The primary challenge in parameter synchronization is managing the balance between potential training quality loss or reduced convergence speed due to work-

ers training on outdated neural network models and the cost of synchronizing to update these models on the workers. Overall, three main approaches are commonly employed: Synchronous, bounded asynchronous, and asynchronous training.

2.4.1 Fully Synchronized

In synchronous training, after each iteration (processing of a batch), the workers synchronize their parameter updates. This strict synchronization model can be implemented as Bulk Synchronous Parallel (BSP) model [31], which are often available in data analytics platforms like Hadoop/MapReduce [11] and Spark [27]. The advantage of strict synchronization is that it simplifies model convergence. However, it also makes the training process susceptible to the straggler problem, where the slowest worker can significantly slow down all the others. Synchronous training is commonly implemented in various open-source deep learning frameworks, such as TensorFlow [1] and MXNet [5]. It is particularly suitable for parallel training on a single multi-GPU compute node, where communication delays are minimal, and the computational load is balanced, reducing the impact of the straggler problem.

2.4.2 Asynchronized

In asynchronous training, workers update their models independently without any synchronization, meaning that there's no guarantee on how current or outdated a worker's model might be. Consequently, it becomes challenging to mathematically reason about model convergence. However, this approach offers the most flexibility to workers in their training process, completely avoiding any straggler problems.

Similar to synchronous training, asynchronous training is well-developed, with multiple implementations in current open-source machine learning frameworks like TensorFlow [1],

MXNet [5], and PyTorch. Asynchronous training takes advantage of the approximate nature of neural network training. Neural network models aim to approximate target functions as closely as possible, allowing for small deviations and non-determinism in the training process without significantly affecting model accuracy. This differs from "strict" problems in data analytics, such as database queries, where deterministic results are essential.

2.4.3 Partially Synchronized

In partially synchronized or bounded asynchronous training, workers may train on somewhat outdated parameters, but this staleness is limited and can be bounded [7]. Bounded staleness permits mathematical analysis and proof of model convergence properties. This bound provides workers with more independence to make training progress, helping mitigate the straggler problem to some extent and increasing overall throughput.

The partial or bounded synchronous model isn't widely implemented in neural network frameworks. This is because it has been observed that the benefits of partial synchronous were not significant enough in frameworks like Tensorflow or PyTorch [39], where the delays were small due to the consistent performance of GPU-intensive operations.

However, the concept of partial synchronization becomes more relevant when introducing heterogeneity into a distributed system. In heterogeneous environments, where workers may have varying computational capabilities or network speeds, achieving balanced progress in a fully asynchronous setup can be challenging. Partial synchronization, with its controlled staleness, can be a valuable strategy to maintain reasonable training efficiency while accommodating differences in worker performance.

2.5 Fine Grain Model Parallelism

Fine-grained parallelism involves breaking down a program into many small tasks and assigning these tasks to numerous processors. Each task does a small amount of work, and this work is evenly distributed among the processors, which helps with load balancing. However, because each task processes a small amount of data, a large number of processors are needed to complete the work. This leads to increased communication and synchronization overhead. Fine-grained parallelism works best in architectures that support fast communication, particularly in shared memory systems with low communication overhead [15]. Detecting fine-grained parallelism in a program is challenging for programmers, so it often falls to compilers to identify it.

The granularity, or the size of the tasks, affects the performance of parallel computers. Using fine-grained tasks can increase parallelism and speed up processing, but it can also introduce synchronization and scheduling challenges. To reduce communication overhead, the granularity can be increased, resulting in coarse-grained tasks, which have less communication overhead but may lead to load imbalance. Optimal performance is typically found between the two extremes of fine-grained and coarse-grained parallelism. Determining the best granularity for parallel processing depends on various factors and varies from problem to problem. Researchers have proposed different solutions to help find the right granularity for specific applications. [14]

Zeng et al. [38], have applied a Codelet Model, which is a fine grain dataflow-based execution model. They group 10-15 neurons into a codelet, and this approach has resulted in a 60% higher speedup compared to OpenMP, a commonly used coarse-grain multithreaded parallel computing model. They conducted this test on LeNet-5, a 7-layer convolutional neural network, using a single-node computing system. By integrating this fine-grain model with a distributed computing system, a compounded speedup can be anticipated that leverages

the advantages of both the fine-grain model and the distributed system. This could lead to even more significant performance improvements.

2.6 Summary

This area of research has seen significant progress, with many optimization techniques developed to address the challenges in neural network training. We've examined various common methods proposed by researchers, including the analysis of distributed infrastructures, techniques for parallelization, scheduling, and data management. A wide range of scalable deep learning techniques is now available in open-source frameworks.

However, a noteworthy observation is that many of these research efforts have not adequately tackled the issue of system heterogeneity. To effectively manage heterogeneous systems, new infrastructures and tools are essential. These tools should enable us to make the most of the diverse hardware and software resources available. As the infrastructure becomes increasingly diverse, machine learning tools should not merely cope with this heterogeneity but also harness it to optimize the training process.

Chapter 3

AllReduce Local Optimization

In the context of parallel and distributed computing, AllReduce is an algorithm where every process or worker shares its data with all other processes, and a reduction operation (e.g., sum, multiplication, max, min) is applied. The primary goal is to condense the target arrays in all processes or workers into a single array, and the result is then distributed or broadcasted back to all processes.

The AllReduce operation can be implemented using various algorithms, and the choice of algorithm often depends on the specific distributed structure or communication pattern of the system. Different algorithms may have varying performance characteristics based on factors such as the number of processes, network topology, and available communication bandwidth. The choice of algorithm depends on the specific characteristics and constraints of the distributed system. Optimizing AllReduce is crucial for achieving good performance in large-scale parallel and distributed computing environments.

3.1 AllReduce Algorithms and Comparison

Some common algorithms for AllReduce include:

1. Scatter-Gather AllReduce: Involves a scatter phase where data is sent to different processes, followed by a gather phase to combine the results.

2. Ring-based AllReduce: Processes form a logical ring, and data is passed along the ring through a series of point-to-point communications until all processes have the final result.

3. Recursive Halving/Doubling AllReduce: a tree-based approach, but organized in a recursive manner. It's particularly efficient when the number of processes is a power of two.

4. Linear AllReduce: Uses a linear communication pattern to efficiently combine data across processes.

We will examine and compare these algorithms in the following sections.

3.1.1 Centralized AllReduce

As shown in Fig 3.1, there is one centralized driver, and all the other workers send all the elements to the driver, after the driver process the reduce operator, it will send the results back to the other processes.

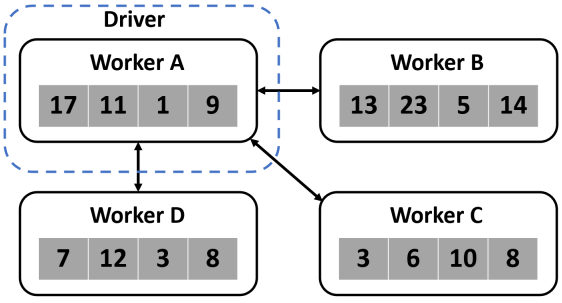


Figure 3.1: Centralized AllReduce

Suppose we have P processors, and each of them has N element. We wanted to calculate the elements sent across the network. In step 1, every worker sends its copy the the driver, so N

elements will be sent $(P-1)$ times. In step 2, after the driver apply the reduce operator and send the results back to the other processes, another $N(P-1)$ elements across the network. So the total number of element sent across the network during this centralized AllReduce is $2(N \times (P-1))$

3.1.2 Ring AllReduce

Assume we have P number of the processes, and each with some data in an array. These processes are organized in a ring. Here's how they collaborate using a Ring-AllReduce algorithm as shown in the Figure 3.2, and the algorithm can be describe as follow:

1. Each process divides its array into chunks. Let's call each chunk $chunk[p]$, where p is the process number.
2. Each process sends its chunk to the next process and receives a chunk from the previous process simultaneously.
3. The process then combines the received chunk with its own chunk using a reduction operation (like adding them together) and sends the result to the next process.
4. This process repeats $P-1$ times, so each process gets a different portion of the final result.
5. After going through all processes, every chunk has traveled around the ring, and each process holds a portion of the final result.
6. To complete the AllReduce operation, processes share their distributed partial results without doing additional reduction operations.

Compare the amount of communication of Ring-AllReduce to that of the Centralized algorithm we mentioned in earlier section above. In the Ring-AllReduce algorithm, we can calculate the amount of communication in each process in the following way. In the earlier

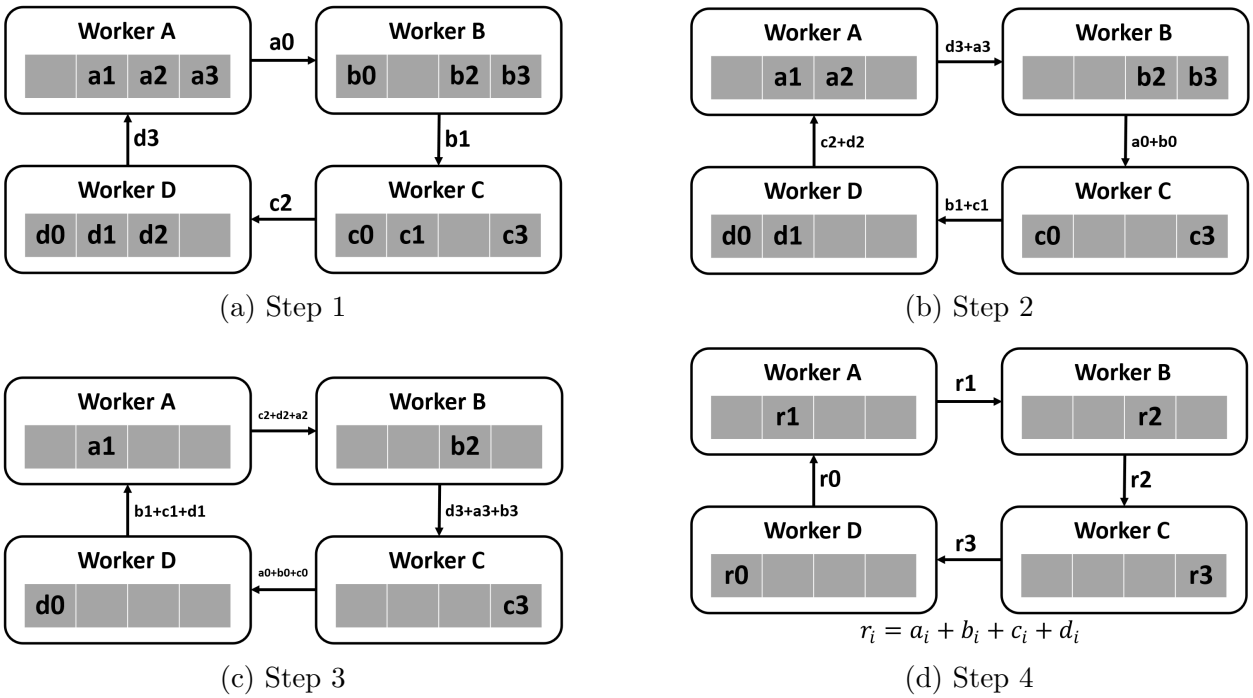


Figure 3.2: Four Steps of Ring AllReduce

half of the algorithm, each process sends an array, the size of which is N/P and $P-1$ times. Next, each process again sends an array of the same size $P-1$ times. The total amount of data each process sends throughout the algorithm is $2N(P-1)/P$, which is practically independent of P. Thus, the Ring-AllReduce algorithm is more efficient than the simple algorithm because it eliminates the bottleneck process by distributing computation and communication evenly over all participant processes. Many AllReduce implementations adopt Ring-AllReduce, and it is suitable for distributed deep learning workloads as well.

3.1.3 Other AllReduce Algorithms

Recursive Doubling AllReduce is a tree-based approach, and organized in a recursive manner. It's particularly efficient when the number of processes is a power of two as shown in Figure 3.3. Processes are paired up in a binary tree structure. In each step, each process exchanges data with its paired partner. The exchanged data is combined using a reduction operation

(e.g., sum, max, min). The binary tree is recursively traversed, and in each step, processes exchange data and perform the reduction operation. This continues until all processes have combined their local data. The amount of data sent will be proportional to $logP$.

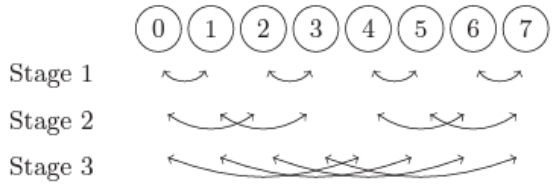


Figure 3.3: Recursive Doubling AllReduce

Another AllReduce algorithm is linear AllReduce, which means the data is passed from one node to another in a linear direction.In each step, processes send their local data to a designated next process. The designated process combines the received data with its own using a reduction operation (e.g., sum, max, min). The combined result is then broadcasted back to all processes. This linear communication process is repeated a number of times until all processes have combined their local data. The amount of data sent will be proportional to the number of processors in the sequence P

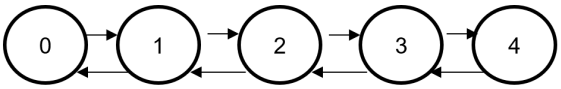


Figure 3.4: Linear AllReduce

3.1.4 AllReduce Performance on Single Node

The memory hierarchy plays a critical role in influencing the performance of computer architectural design, algorithm predictions, and low-level programming constructs. It particularly impacts the concept of "locality of reference." Locality of reference refers to the tendency of a computer program to access a relatively small portion of its memory space frequently for a specific period, whether in terms of reading or writing data. Optimizing memory access patterns to maximize locality of reference is crucial for improving the performance of

computer systems. It involves considerations related to cache management, data layout, and algorithm design. Understanding and exploiting locality of reference is essential for efficient memory utilization and overall system performance.

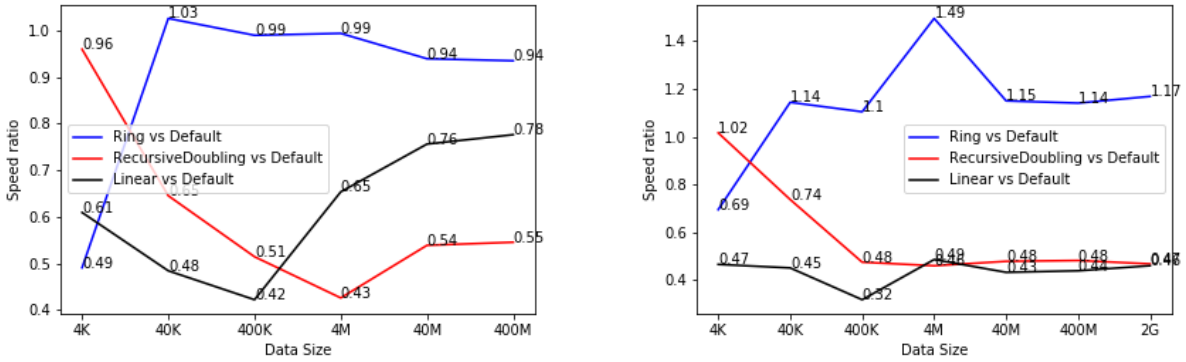
We have performed a single node test of four different AllReduce algorithms and compare their performance by changing the input data size. The idea is trying to find the relationship of memory hierarchy and optimal data size. As we mentioned in Chapter 2.5 fine grain parallelism, choosing the best granularity directly affects the overall performance. The test node we used are listed in table 4.2

Table 3.1: Hardware Configurations of Single Node Allreduce

Node	***CPU***	***Cores***	***L1 Cache***	***L2 Cache***	***L3 Cache***
node1	Ryzen 3 1600 3.6GHz	6	96KB/core	512KB/core	16MB shared
node2	Xeon(R) W-2155 3.3GHz	10	32KB/core	1024KB/core	14MB shared

In Figure 3.5, it's observed that both centralized AllReduce and ring AllReduce exhibit better performance compared to linear and recursive doubling as the data size increases. Both centralized and ring AllReduce show similar and improved performance as the amount of data grows. In the case of ring AllReduce, there's an optimal data size for the best performance. This means that as the data size increases, performance improves, but only up to a certain point. Beyond this optimal data size (specifically, at $4M$ in Figure 3.5 (b)), ring AllReduce outperforms centralized AllReduce by 1.5 times.

The tests of different reduce operator behaviors on the performance of these algorithms show consistent patterns across different reduce operators, as shown in 3.6. Regardless of the specific reduce operator used, the performance patterns of the algorithms remained the same. The shape of the performance curves was consistent for the same hardware, indicating a definite relative performance among the four algorithms. This consistency is advantageous



(a) Single Node 6 Cores

(b) Single Node 10 Cores

Figure 3.5: AllReduce Algorithms Performance on Single Node

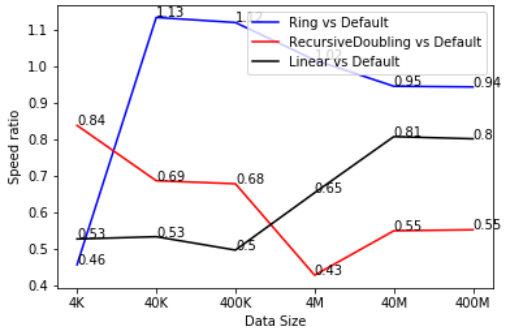
because it allows for the identification of the most suitable algorithm and optimal data size for a given case. It means that users can confidently choose the appropriate algorithm and data size before integrating these variables into a large distributed system.

In summary, having a clear understanding of how different algorithms perform under various conditions, including different reduce operators, provides valuable insights for making informed choices in the design and integration of large distributed systems.

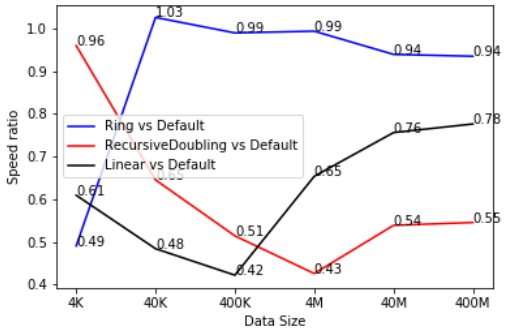
3.1.5 Memory Effect and Local Optimization

We would like to examine the cache effect on the AllReduce operators, and this is directly related with the granularity of fine grain parallelism. We designed this experiments as shown in Figure 3.8. Fig 3.8 (a) shows core 0, core 1, and core 2 have independent L1 and L2 cache, and shared L3 cache. This setup allows the examination of how the shared L3 cache impacts the performance of AllReduce operators. The design is intended to explore how caching mechanisms, particularly at the L3 level, influence the parallel execution of AllReduce operations on different cores.

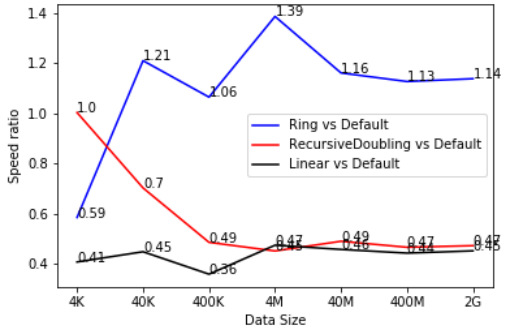
The observation indicates a clear performance improvement for three cores with shared L3



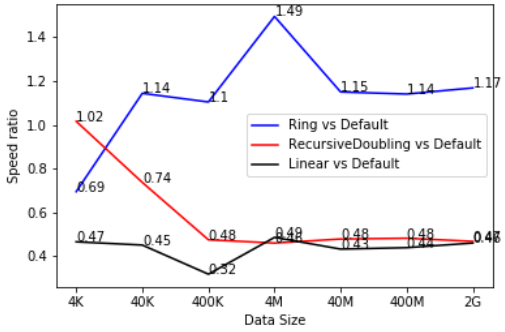
(a) 6 Cores SUM Operator



(b) 6 Cores PROD Operator

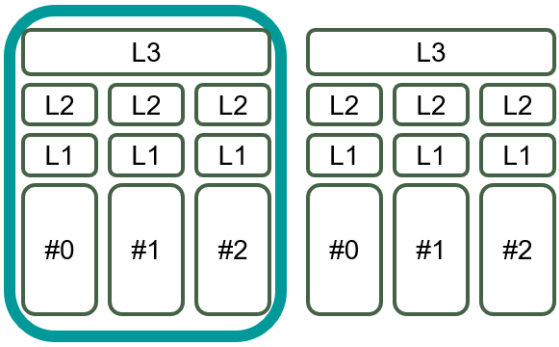


(c) 10 Cores SUM Operator

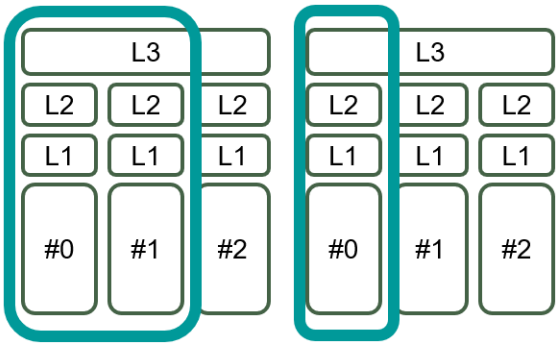


(d) 10 Cores PROD Operator

Figure 3.6: AllReduce SUM and PROD Operator Comparison on Single Node



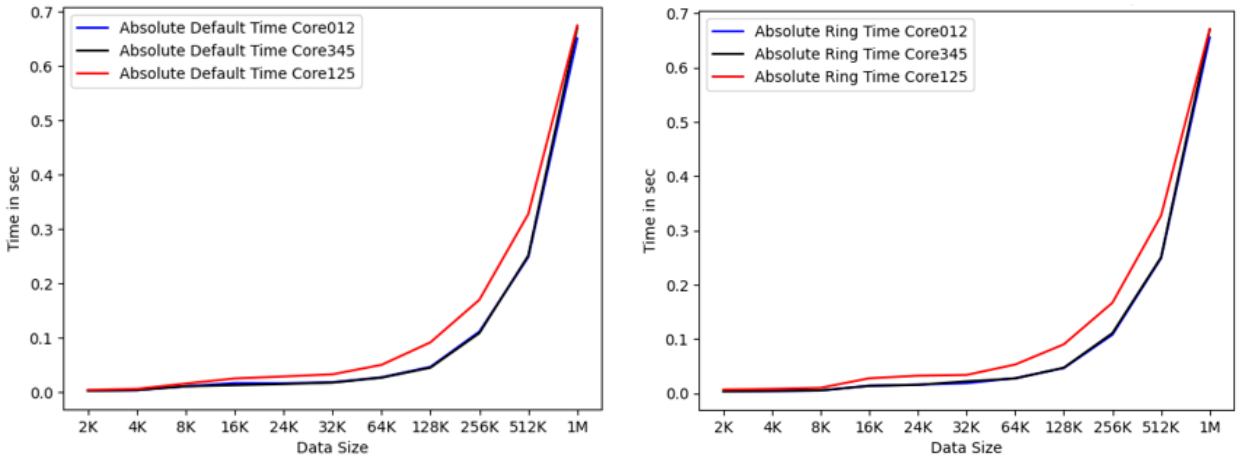
(a) Three Cores with Shared L3 Cache



(b) Three Cores without Shared L3 Cache

Figure 3.7: Cache and Granularity on Allreduce Operations

cache, especially in the data size range from 32K to 512K (where 512K is the size of L2 cache in Node 2 as shown in Table 4.2). The key takeaway is that constraining the data size within the size of the L2 cache leads to better performance. Key points from the observation: First, we observed L3 Cache impact. Three cores sharing an L3 cache exhibit better performance, highlighting the importance of cache effects on parallel operations. Second, we observed optimal data size. The performance improvement is particularly visible when the data size is within the size of the L2 cache in Node 2 (512K). This suggests that optimizing data size within the available cache capacity is crucial for achieving better performance. Third, we observed similar shapes for centralized and ring AllReduce: Both centralized and ring AllReduce operations generate similar and almost identical plot shapes, indicating a consistent performance pattern across these two algorithms.



(a) Centralized Allreduce SUM Operator (b) Ring Allreduce SUM Operator

Figure 3.8: Cache and Granularity on Allreduce Operations

In summary, optimizing data size based on cache constraints is a key factor for achieving better performance. Additionally, the similar performance shapes for centralized and ring AllReduce suggest that the choice between these two algorithms may depend on other considerations, given their comparable performance in this context.

The investigation into the performance of ring AllReduce versus centralized AllReduce continues with a focus on the speed ratio between these two algorithms under shared L3 and

non-shared L3 cache conditions. From Figure 3.9, in both shared and non-shared L3 cache scenarios, ring AllReduce exhibits an optimal data size that results in the best speedup. Under shared L3 cache conditions, the speed ratio between ring and centralized AllReduce can be as high as 2.2 times for a data size of 8K.in the case of non-shared L3 cache, the speed ratio reaches up to 1.65 times for an optimal data size. This indicates that the benefits of using ring AllReduce persist. In summary, the comparison between ring and centralized AllReduce, considering speed ratios, highlights the importance of data size optimization for achieving optimal speedup. The significant speedup observed, particularly in the shared L3 cache case, emphasizes the impact of cache configurations on the performance of parallel algorithms.

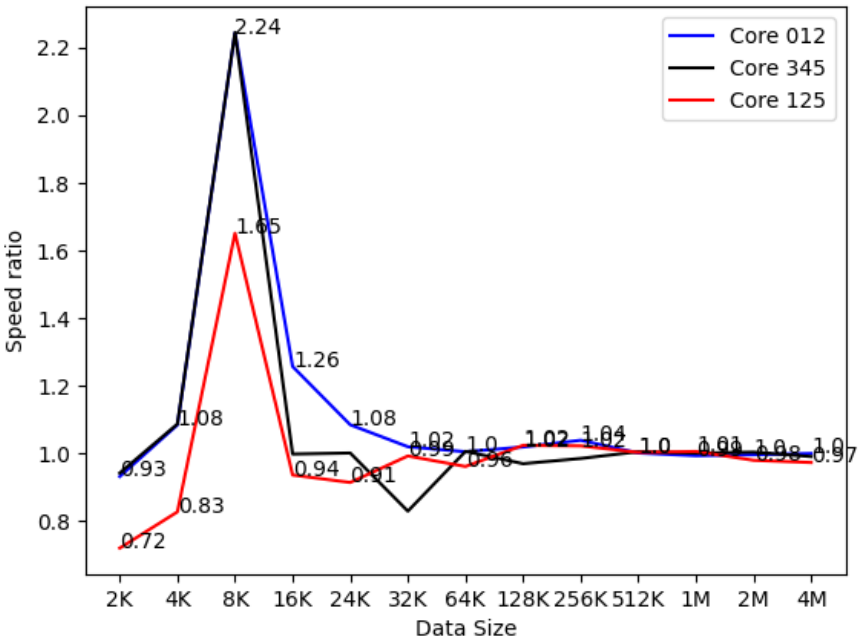


Figure 3.9: Speed Ratio of Ring Allreduce vs Centralized Allreduce

The observation of improved performance with a shared L3 cache raises the question of whether sub-grouping, where cores are separated based on cache sharing, can lead to better performance. Specifically, if cores 0, 1, and 2 are grouped together, and 3, 4, and 5 are in another group, the comparison is made to determine which structure yields better performance. The structure is illustrated in Figure 3.10, with cores grouped into sub-groups.

The main question is which sub-grouping structure can achieve better performance. The exploration of sub-grouping aims to understand the impact of how cores are grouped on the overall performance of parallel algorithms. The experiment will give an example on whether certain sub-groupings can outperform others in specific scenarios.

(a) Without Subgroups (b) With Subgroups

Figure 3.10: Cache and Granularity on Allreduce Operations

The results, as depicted in Figure 3.11, indicate that, for the majority of data sizes in this experiment, sticking with a single group yields better performance compared to having two subgroups. Only when the data size is smaller than 4K, the performance with two groups is slightly better than the case with one group. The observation suggests that, for most of the data sizes considered in this experiment, the optimal choice is to stick with a single group.

In summary, optimizing AllReduce operations in shared memory conditions,involves considering cache and memory effects, highlighting the importance of identifying an optimal data size for fine-grain parallelism. The observed fast access in shared memory scenarios suggests that subgrouping or other logical structures may be negligible. However, the question of when to consider subgroups and the specific conditions favoring their use remains open. The next section will undertake a mathematical analysis to explore the relationship between computation and data transfer in distributed systems, aiming to provide nuanced insights into decision-making for performance optimization in such environments.

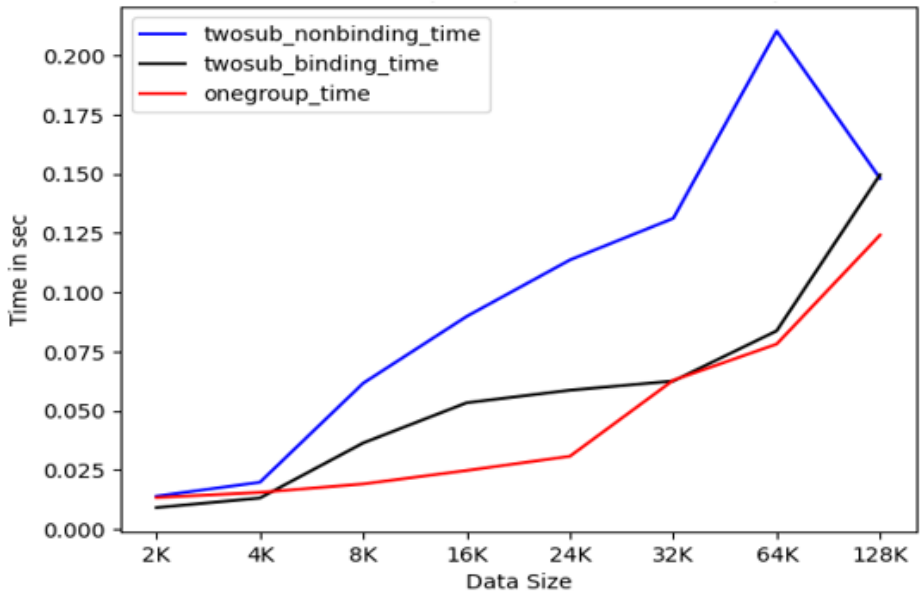


Figure 3.11: Sub Groups and One Group Comparison

3.2 Mathematical Analysis of Compute and Transfer Ratio in Distributed System Decisions

In this section, we use mathematical abstractions to simplify representations. Uppercase letters are used to represent node names, and the numbers on each node represent the unit cost (cost per byte) associated with a specific operation, considering the workload and computational power. The numbers on the lines connecting nodes indicate the transfer cost per byte for the respective connections. A job is symbolized by a square, incorporating a job name, operation, and the size of the processed data. Taking Figure 3.12 as an example, it illustrates an operation denoted as op_a with a data size of n. The cost is 1 unit per byte if executed on node A, 10 units per byte if on nodes B or C. When transferring between nodes AB or AC, the cost is 100 units per byte, and it becomes 1000 units per byte between nodes BC.

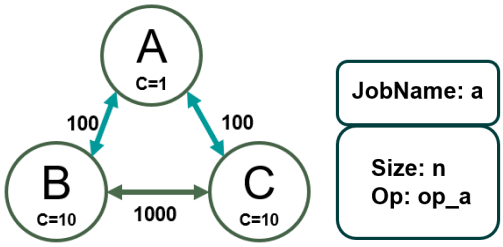


Figure 3.12: Abstraction of Nodes and Costs

3.2.1 Three Nodes Scenario with Single Operation

In the case of three nodes, suppose we have the cost on A equals 1 unit per byte, and cost on B and C is 10 units per byte. We would like to understand what's the relationship on the network transfer between A and B, or A and C and the cost on each node. So we assume cost X on the network transfer of AB and AC. Now we have three options to run the this job: sequential on fast node A, include one of the slow node B or C, or include all the three nodes ABC. Following is the analysis to determine when should choose one of these options. Sequential on Node A:

$$cost_A = n \cdot 1 = n \tag{3.1}$$

Parallel on Node AB or AC:

$$cost_{AB} = cost_{compute} + cost_{transfer} = \frac{10}{11} \cdot n \cdot 1 + \frac{1}{11} \cdot n \cdot X \tag{3.2}$$

If we want to include B or C into the operation, we should let $cost_{AB} < cost_A$, and that gives $X < 1$, and it means that the cost on the network should be smaller than cost on A.

Parallel on Node ABC:

$$cost_{ABC} = cost_{compute} + cost_{transfer} = \frac{10}{12} \cdot n \cdot 1 + \frac{1}{12} \cdot n \cdot X \tag{3.3}$$

If we want to include both B and C into the operation, we should let $cost_{ABC} < cost_A$, and that gives $X < 2$, and it means that the cost on the network should be less than twice of cost on A.

We can conclude that the relationship between X and ratio of cost on A and B determines if B and C should join into the whole operation.

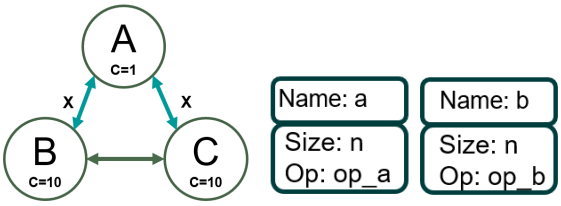


Figure 3.13: Mathematical Analysis of Three Nodes Multiple Operations

3.2.2 Three Nodes Scenario with Multiple Operations

Suppose we have two operations op_A and op_B, and define a ratio of cost on a certain node as:

$$\frac{op_b}{op_a} = r, r > 1 \tag{3.4}$$

Sequential on Node A:

$$cost_A = n \cdot (r + 1) = n \tag{3.5}$$

Parallel on Node ABC, with cost on A:

$$cost_A = n \cdot r = nr \tag{3.6}$$

Cost on B or C:

$$cost_{B/C} = 10 \cdot \frac{n}{2} + X \cdot \frac{n}{2} = (10 + X) \cdot \frac{n}{2} \tag{3.7}$$

Let $cost_{B/C} < cost_A$, we can get:

$$cost_{B/C} = (10 + X) \cdot \frac{n}{2} < (r + 1) \cdot n \tag{3.8}$$

and continue to solve the equation, we have:

$$r >= 4 + \frac{X}{2} < (r + 1) \cdot n \tag{3.9}$$

We can examine the equation, and the value 4 comes from the compute power ratio of Node B and Node A, or Node C and Node A. The value 2 comes from the number of worker nodes. These two numbers should be changed back to variables to extend this equation to general case.

In the two examples provided earlier, we showcased an abstraction model designed to depict a heterogeneous environment characterized by uneven computational power and disparate network transfer rates. In the upcoming subsection, we will broaden the scope to encompass N nodes and delve into the criteria for deciding when to utilize multiple nodes within such a heterogeneous environment.

3.2.3 N Nodes Scenario

As shown in Fig 3.14, We use the $bigO$ notation to represent the unit cost (cost per byte) on a certain operation based on the workload and compute power. The node with $O(n)$ in the circle means it will cost n unit of time to complete a certain operation with a certain data with a size of n. All the nodes in the system can be separated into p number of groups(vertically) from M_1 to M_p. We characterized the heterogeneous node with same order of magnitude into the same homogeneous group. Grouping algorithms can be used to separate the nodes first by transfer cost among the nodes and the compute difference, The algorithms are shown below:

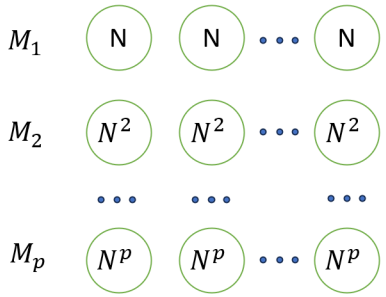


Figure 3.14: Mathematical Analysis of Nodes Groups

As discussed in the related work, the recent hybrid cluster system is trying to utilize the ring allreduce with synchronization,and only discovers the homogeneous part of a cluster. And the cluster is composed of large number of homogeneous node. We are proposing instead of just discover the homogeneous part, but we should also create the homogeneous group which composed of the heterogeneous node. In the cases of having more heterogeneous node and fewer homogeneous nodes in the system, our framework is able to compose the heterogeneous nodes into homogeneous group by considering their computer power and transfer rate between them. In this way, the allreduce and fully synchronization can still be maintained in the system, and also all the compute nodes in the system can be utilized.

Mathematical analysis of compute and transfer ratio in system decisions

Once we have the grouping and characteristics are done, mathematical analysis can be applied on the static structure to give an early decision on how many nodes in the system should be used, and how they should synchronize with each other when a computation job is given. Using the setup in Fig 3.14, we consider the transfer cost to be identical among each of the nodes suppose we have done the separation by transfer cost beforehand, using $O(T_r)$. We also define $O(r_b N)$ where $r_b = op_b/op_a$, which means the ratio of the cost for different operation op_a and op_b on the same compute node with same data size. Even op_a and op_b are within the same order of magnitude, but they are still different by a ratio of r_b.

Assume we have m number of jobs $(a_1, a_2, ..., a_m)$ with the same workload for simplicity, which is also the case of data parallelism strategy used in distributed deep neural network training where parallel compute nodes are used to process the partitioned training dataset using a replica of the deep learning model. If $m < M_1$ we have two options: either serial computation on one of the node in M_1, or parallel within the group of M_1. The cost for the serial option will be mN, and the parallel on M_1 will cost nT_r for the transfer data and N for the compute. So the total cost is $nT_r + N$ when jobs are averaged assigned onto M_1 group. If the number of jobs is larger than the number of node in M_1, i.e. $m > M_1$, we have the options to either send batch (m/M_1) jobs onto M_1,

$$parallel_{M1} = (nT_r + N)(m/M_1 + 1) \tag{3.10}$$

or utilize both M_1 and M_2 groups with M_1 jobs are sent to M_1 and $(m - M1)$ jobs are sent to M_2, we can have

$$parallel_{M_1M_2} = (nT_r + N)(m/(NM_1 + M_2)) + (nTr_r + N^2)(m/(NM_1 + M_2)) \tag{3.11}$$

$$= (2nT_r + N + N^2)(m/(N^2M_1 + NM_2 + M_3)) \tag{3.12}$$

Apply the same logic on M_3 when all the $M_1, M_2 and M_3$ are used, so there are $(m - M_1 - nM_2/N)$ jobs on M_3, and total cost on this situation will be:

$$parallel_{M_1M_2M_3} = (nT_r + N + nT_r + N^2)(m/(N^2M_1 + NM_2 + M_3)) \tag{3.13}$$

$$= (3nT_r + N + N^2 + N^3)(m/(N^2M_1 + NM_2 + M_3)) \tag{3.14}$$

by deduction we can conclude that:

$$parallel_{M_1M_2M_3...M_p} = (pnT_r + N + N^2 + N^3 + ... + N^p)(m/(N^{p-1}M_1 + N^{p-2}M_2 + N^{p-3}M_3 + ... + M_p)) \tag{3.15}$$

In the overall analysis, we have a set of p equations to evaluate, allowing us to compare and select the option with the least cost. This approach serves as a mathematical static strategy, laying the groundwork before transitioning to the runtime layer. By evaluating costs across different options, we can make informed decisions, contributing to the efficient management and optimization of distributed system performance.

Chapter 4

Hierarchical Heterogeneous Framework, Simulation, and Experiment

The overall structure of the framework tool we are proposing utilizes flexible synchronization and hybrid architecture which can adaptively and dynamically adjust the architecture based on the performance of heterogeneous hardware and different algorithms. The targeted platform includes heterogeneous cores, heterogeneous node with different computation capability, distributed nodes with heterogeneous network environment. The overall hierarchical structure includes: First, a number of nodes form a unit group. The criteria of which nodes are grouped and how many nodes are in one group can be heuristic for the static structure. For example, heuristically we would like the identical nodes to be in one group and jobs can be evenly split among the nodes in the group. However sometimes we would like to have less powerful nodes to work together with powerful nodes on some jobs which requires less computing power in a way to fully utilize the resources.

Consider the nodes communication within each group, either centralized or decentralized way can be applied based on the characteristics of the nodes within the group. Each lower level group can also be considered a member of the high level group. The main issue we have discovered in the recent research paper is that the scalability issue is not considered. Many of the algorithms and strategies are sensitive to the number of nodes in the group. So in order to organize large amount of resources, hierarchical structure has to be applied into the system.

The algorithms related and specific hardware to design the initial structure and dynamic structure structure, aiming to use different type of algorithms based on the profile of jobs to be finished. For computing bound, memory bound and communication bound, we design the corresponding strategies, which will be described in section IV.

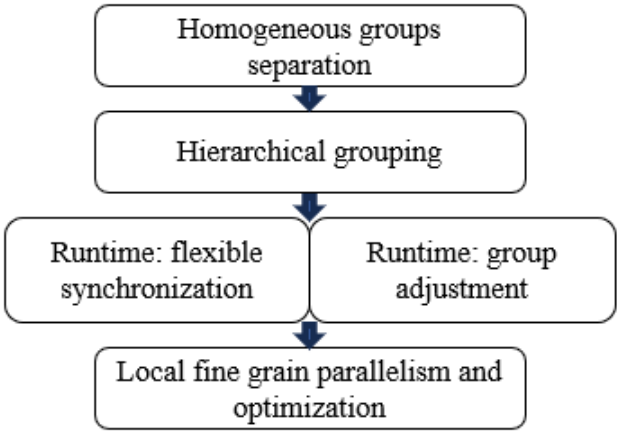


Figure 4.1: Proposed Framework Layers

First our framework tries to group the heterogeneous nodes, and forms homogeneous group and in this way we can utilize the ring AllReduce and synchronization. Second level utilize the hierarchical grouping which is the way to apply scalable features. The number of node within a group has to be limited, and prevent the group size to be too large. On next level, consider the rate of compute and transfer to adjust the grouping. It should be avoided that lower transfer rate node is grouped with high transfer rate node. At last, utilize the fine grain parallelism on each compute node to fully discover compute efficiency and the relationship

with local memory structure. Each level of the framework will be discussed in the following subsections.

4.1 Homogeneous Group Separation

As discussed in the related work, the recent hybrid cluster system is trying to utilize the ring AllReduce with synchronization,and only discovers the homogeneous part of a cluster. And the cluster is composed of large number of homogeneous node. We are proposing instead of just discover the homogeneous part, but we should also create the homogeneous group which composed of the heterogeneous node. In the cases of having more heterogeneous node and fewer homogeneous nodes in the system, our framework is able to compose the heterogeneous nodes into homogeneous group by considering their computer power and transfer rate between them. In this way, the AllReduce and fully synchronization can still be maintained in the system, and also all the compute nodes in the system can be utilized.

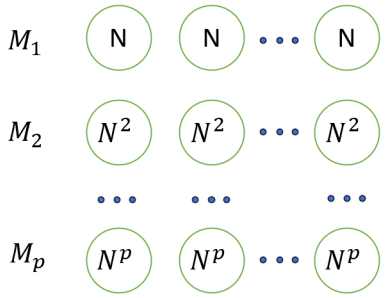


Figure 4.2: Homogeneous group separation

As shown in Fig 2, We use the *bigO* notation to represent the unit cost (cost per byte) on a certain operation based on the workload and compute power. The node with $O(n)$ in the circle means it will cost n unit of time to complete a certain operation with a certain data with a size of n. All the nodes in the system can be separated into p number of groups(vertically) from M_1 to M_p. We characterized the heterogeneous node with same order of magnitude into the same homogeneous group. Grouping algorithms can be used to separate the nodes

first by transfer cost among the nodes and the compute difference, The algorithms are shown below:

Algorithm 1 Homogeneous Group Separation Algorithm

W_i represent worker;
List of $Group[]$ distinguished by communication cost;
List of $group[]$ distinguished by compute power;
for $i = 0$ **to** n **do**
  for $j = i - 1$ **to** 1 **do**
    if $O(C_{i,i-1}) = O(C_{j,j-1})$ **then**
      $W_i.groupId = W_{c_j}.groupId$;
      $Group[W_i.groupId].add(W_i)$;
    end if
  end for
end for
return $Group[], group[]$

4.2 Hierarchical Structure

It is generally accepted that the ring-AllReduce theoretically is faster comparing with the parameter server with used centralized architecture, assuming all the compute node on the ring has the finish the same workload and communicate cost is same between nodes on the ring.

One problem in the cluster system is scalability, and even if the ring AllReduce and fully synchronization architecture still have the concern on the scalability issue. The recent primitive experiment shows that with the increasingly number of nodes, the benefit of using ring AllReduce rather than parameter server decreased. It can be explained using the probability of including a node of slow transfer rate into the whole ring structure has increased, so the overall performance of reduced. and also more node included into the ring, we have more chance of having a faulty node in the whole structure. So we should consider improving the scalability using the hierarchical way of grouping, to limit the size of the ring. So that

drives us to explore the hierarchical solutions if the nodes are heterogeneous and the number of nodes are increasing. Assume we have n number of compute nodes, and n is a large number. The intuitively straight way to form a hierarchy will be take the square root of n, so the total number of the groups will be $\sqrt{n}$ and the size of each group is also $\sqrt{n}$. For each group, the hierarchical formation can still continue to $\sqrt[2]{n}$. Fig 3 shows two examples of hierarchical structures with logical connections to form different structure, which include the ring-AllReduce configuration and parameter configuration.

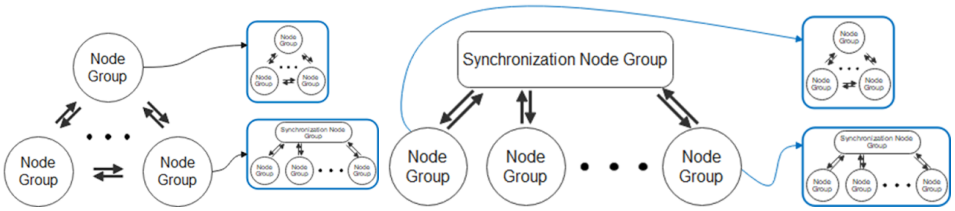


Figure 4.3: Examples of hierarchical logical structures

When choosing which nodes should be formed in one subgroup, random formation is one strategy if no particular feature can be discovered among the nodes. The Central limit theorem tells us as we take more samples, especially large ones, the graph of the sample means will look more like a normal distribution. So we can make sure that the overall compute power will be a normal distribution.

4.3 Simulation with Fluctuate Computation and Communication Cost

To analyze the impact of fluctuating computation and communication costs on training performance, we've crafted a simulation. This simulation is designed to replicate scenarios related to real-world conditions where computational resources and communication bandwidth may vary. In this simulated environment, we introduce controlled variations in computation power and communication speed, mimicking the challenges encountered in dynamic comput-

ing setups. Our focus is on understanding how these fluctuations influence critical training metrics, including convergence rates, accuracy, and resource utilization.

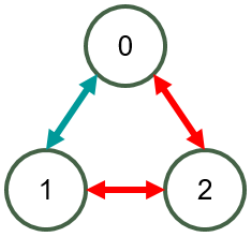


Figure 4.4: Three Nodes Unstable Network

As shown in Figure 4.4, Let's use the example of three nodes (Node 0, Node 1, and Node 2) connected to each other. Nodes 0 and 1 have fast connections, while Node 2 has a slow connection with Node 0 and 1. A $wait\_function$ time will be inserted into the code to simulation the slow connection. We'll then observe the behavior of the unstable network connection in relation to the training performance, and monitor and record relevant performance metrics such as training loss, accuracy, and convergence time.

4.3.1 Fixed Delay

The outcome is depicted in Figure 4.5. We conducted two distinct tests with convergence thresholds set at $loss = 0.1$ and $loss = 0.05$ using MNIST dataset and modified LeNet network. Our aim is to examine how the synchronization rate with Node 2 impacts the training performance, specifically the time it takes to achieve convergence. Theoretically, Node 2 has a slower connection with Node 0 and 1. To minimize overall delay, it is advisable to decrease the synchronization rate with Node 2. On the other hand, we also need Node 2 to contribute into the simulation. So we anticipate identifying an optimal synchronization rate ranging from 0 to 100%. In essence, rather than opting for full synchronization or asynchronization, we seek to determine the optimal rate for synchronization with Node 2.

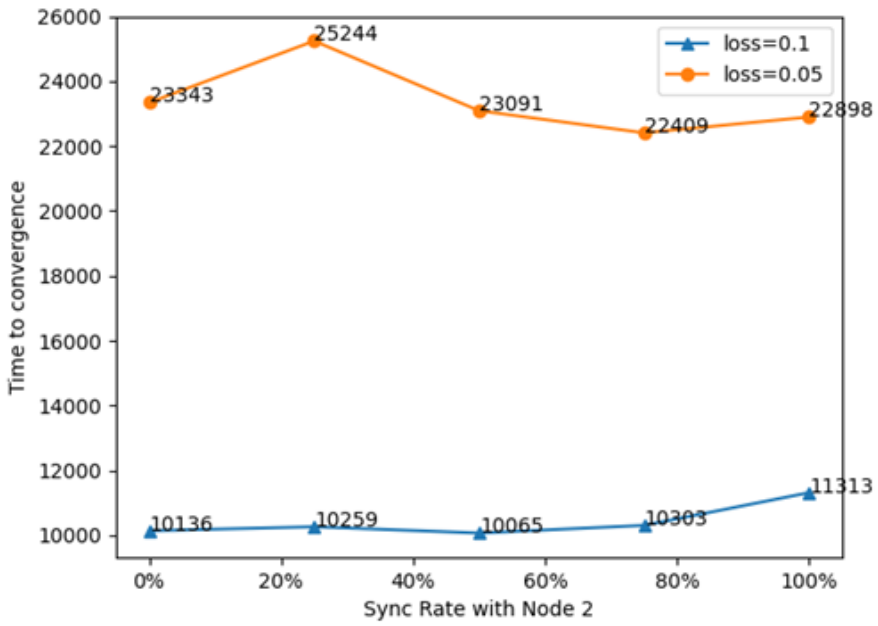


Figure 4.5: Three Nodes Unstable Network Simulation Fixed Delay

4.3.2 Random Delay

In the next simulation, we configure the unstable value to be a randomly assigned value within a specified range. Our objective is to investigate the diverse behaviors that may emerge due to the fixed delay influencing performance. Additionally, we aim to discover how the introduction of random delays changes the optimal synchronization rate with Node 2. The results is shown in Figure 4.6, we can see from Figure 4.6 (A), we can see the optimal synchronization rate with Node 2 reduced to around 10% for using the threshold $loss = 0.1$.

In addition to adjusting the delay factor, we aim to observe the correlation between the synchronization rate and both computation and synchronization times. To achieve this, we introduce two runtime lists, as $run\_time\_list[]$ and $sync\_time\_list[]$, to track the runtime aspects of computation and synchronization. Our goal is to align these values with the synchronization rate. In Figure 4.6 (B), we note that at a 10% optimal synchronization rate, the ratio of synchronization to computation time is approximately 0.02, or 2%. This

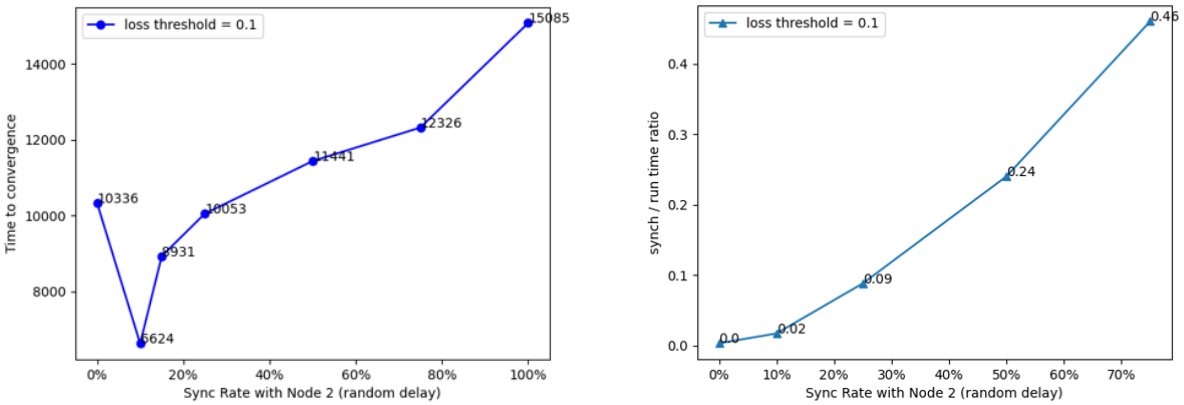


Figure 4.6: Three Nodes Unstable Network Simulation Random Delay

indicates the importance of monitoring the synchronization time relative to computation time, aiming for a 2% ratio during the training process to optimize performance. By adjusting the synchronization rate with Node 2, we can achieve this optimal training performance. If the observed percentage deviates from the 2%, dynamic control of synchronization times with Node 2 becomes necessary.

In summary, our simulations provide evidence that in an unstable computational and synchronization environment or in a heterogeneous environment, employing a flexible synchronization strategy allows for the effective control of the synchronization rate. This control proves valuable in minimizing unnecessary costs associated with synchronization, ultimately enhancing overall system performance. To further enhance these outcomes, it is imperative to get into runtime strategies and explore optimal solutions for this particular context.

4.4 Experiments on Heterogeneous Distributed System

In our study, we implemented and tested our runtime system in three distinct environments: a heterogeneous CPU environment, a heterogeneous GPU environment, and a combined GPU and CPU environment. The evaluation involved a comparison of training performance with various synchronization strategies, including fully synchronized, no synchronization, fixed percentage synchronization, and flexible synchronization. The flexibility in synchronization was achieved by our runtime module, which monitored the runtime behavior of each node and the network conditions. Additionally, we assessed the robustness of our runtime system by introducing delays and variances into the network connections. Notably, our system demonstrated the capability to identify and exclude abnormal nodes from the synchronization process, ensuring the overall performance of the entire system. All tests were implemented and integrated within the PyTorch framework.

4.4.1 Runtime Management Module

Our runtime management module, integrated with compute and transfer variance in a dynamic structure, offers several significant advantages. The logical organization of hierarchical nodes ensures effective utilization, allowing even less powerful nodes to contribute meaningfully to overall system performance. The hierarchical structure enables the dynamic inclusion or removal of nodes from specific groups based on network availability and instantaneous data transfer rates. This adaptability at a local level enables fine-grained configuration adjustments in response to varying computational resources and network conditions. Given the inherent volatility of computational power and network conditions, we have developed a runtime management module. This module serves as a proactive supervisor during neu-

ral network training, capitalizing on flexible synchronization techniques. By adapting to real-time system characteristics, the module optimizes resource allocation, ensuring that the neural network operates efficiently, even in the face of fluctuating computational resources and network conditions.

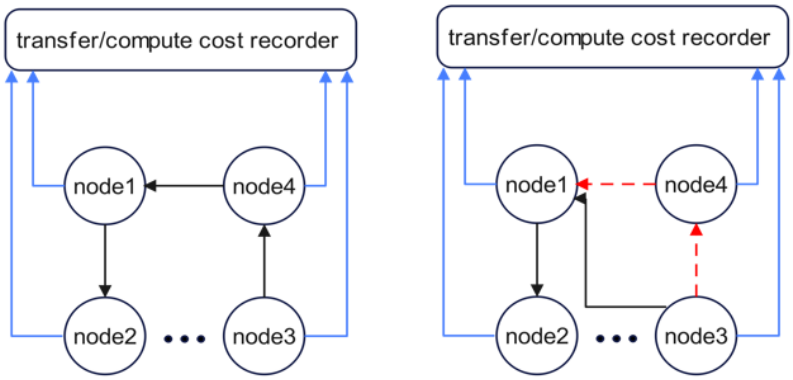


Figure 4.7: Flexible Synchronization

Figure 4.7 shows node 4 is excluded from the ring when the runtime management module detects the abnormal situation from node 4 either because of slowdown in the computation or the congestion happened in the the network connection from node 4 to runtime management module. Node 1, 2 and 3 can form a new ring logically and continue the ring AllReduce process when updating the model parameters.

We can use the abnormal detection as an API access, so further intelligence on the design of the algorithm can be directly plug into the whole system. For example a running average or limited width sliding window algorithm are both used as we develop the runntime system.

4.4.2 Heterogeneous CPU environment

The CPU cluster we formed for evaluation comprises 13 nodes. Nodes with fast connections are put into one group. Only one node in the group serves as communication node with the outside group. The nodes in group 1 and group 2 are within the same magnitude as we use

Algorithm 2 Runtime Distributed Ring AllReduce Algorithm

$node_i$ represents worker i;
$node_i.ring[]$ contains nodes active in ring-AllReduce;
while not converged **do**
  for $i = 0$ **to** n **do**
    if $abnormal\_detection(node_i)$ **then**
      $ring[].remove(node_i)$;
    end if
  end for
  for $i = 0$ **to** n **do**
    $update(node_i.ring[])$;
  end for
end while

Algorithm 3 Runtime Abnormal Detection Algorithm

$node_i$ represents worker i;
$threthold$ represents limit value to define abnormal;
while not converged **do**
  for $i = 0$ **to** n **do**
    if $abs(node_i.cycle\_time - node_i.running\_average) > threthold$ **then**
      $node_i.abnormal = true$;
    end if
    $update(node_i.running\_average)$
  end for
end while

big O notation to describe the compute power. Group 3 and 4 is one more magnitude slow comparing with group 1 and 2.

Table 4.1: Hardware Configurations of Heterogeneous CPUs

Group	***Nodes***	***CPU***	***Network Adapter***
G1	node 0-3	Intel i5	1Gbps
G2	node 4-7	AMD Ryzen 5	1Gbps
G3	node 8-10	Arm V8	1Gbps
G4	node 11-13	Arm V7	1Gbps

We select the LeNet networks and MNIST dataset in this experiment. We record the training time till the test accuracy reaches 95%. In the case of connection all group 1 to 4, the nodes of group 3 and 4 are dropped off during the runtime because the compute power is magnitudes lower and also with a slow connection to group 1 and 2.

Comparing the Fully Synchronization, Percentage Synchronization, and Flexible Synchronization. Percentage Synchronization has close to optimal performance, and flexible synchronization has the advantages on the runtime when network variance is introduced into the computation environment. The plots presented in Figure 4.8 illustrates a significant decrease in the performance of full synchronization when confronted with network or computation variances. While the percentage synchronization also experiences a slowdown, it maintains a relatively high performance by constraining the synchronization rate with slower connections or computation nodes. The runtime flexible synchronization demonstrates an improvement in performance compared to full synchronization, although it falls short of the performance achieved by percentage synchronization.

Despite not matching the efficiency of percentage synchronization, the runtime management module remains effective in addressing the variances or turbulence within the distributed system. This suggests that the runtime flexible synchronization can serve as a valuable tool for mitigating the impact of disruptions in a distributed computing environment.

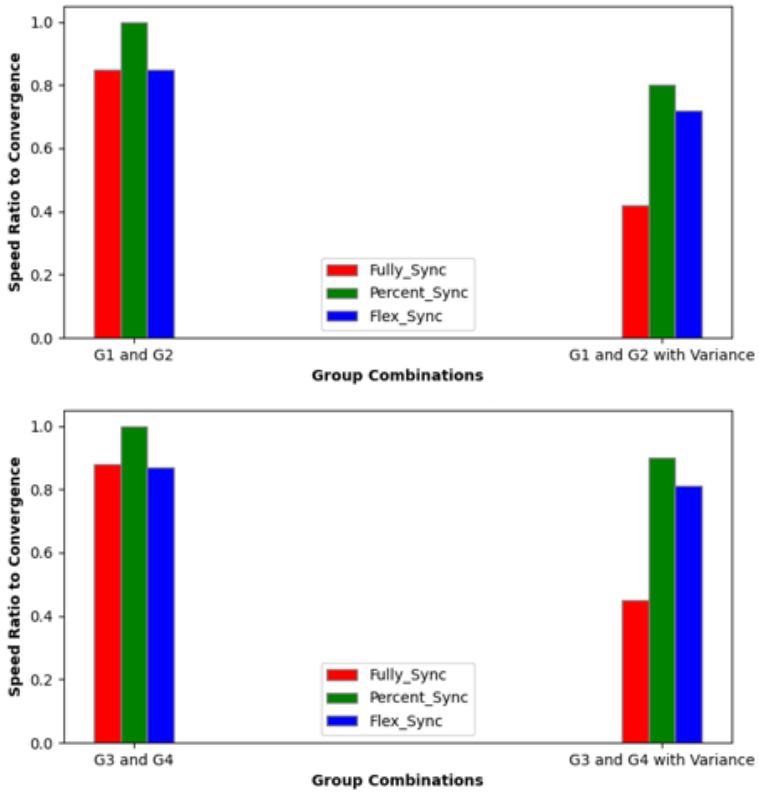


Figure 4.8: Heterogeneous CPU Experiment Result

4.4.3 Heterogeneous GPU environment

For creation of the test in a heterogeneous GPU environment, We use two Nvidia Professional GPUs Quadro P6000 to form one group (G1) in the table, and another Quadro P6000 to form another group (G2) in the table. We choose a commercial GPU Nvidia 3060Ti to be the contrast group with different computation power. The hardware details are listed in Table 4.2.

Table 4.2: Hardware Configurations of Heterogeneous GPUs

Group	***Nodes***	***GPU***	***Network Adapter***
G1	node 0-1	Nvidia P6000	1Gbps
G2	node 2	Nvidia P6000	1Gbps
G3	node 3	Nvidia 3060Ti	1Gbps

We have conducted tests on four distinct sets under varied group conditions. Across the board, the performance of full synchronization shows a notable decline when confronted with huge differences in computer power or substantial variances in the network. This decline was particularly evident in scenarios such as G1 and G2 with network variance or G1 and G3. In contrast, percentage synchronization showcased greater resilience by adeptly regulating the synchronization rate, especially in the presence of slower connections, whether in computation or transfer.

The flexible synchronization, operating as a runtime method to govern the synchronization rate, proved to be a viable alternative. It demonstrated an ability to sustain performance levels relatively well when compared to the more resource-intensive full synchronization. This underscores the effectiveness of adopting a flexible synchronization strategy, particularly in environments characterized by computational disparities or network variations, where it can effectively mitigate unnecessary costs and optimize overall system performance.

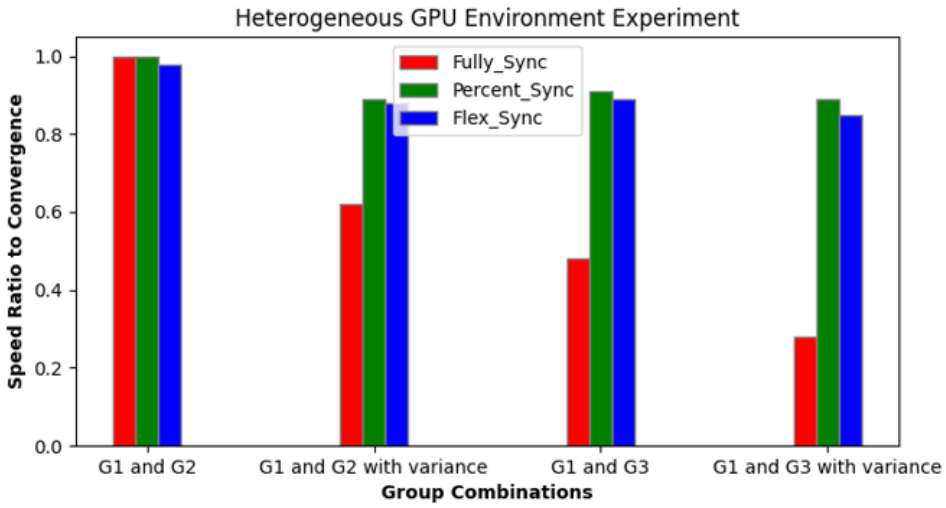


Figure 4.9: Heterogeneous GPU Environment

Chapter 5

Conclusion and Future Work

A dynamically hybrid and hierarchical logical architecture for cluster nodes has been introduced to satisfy the adaptivity and the scalability issues in distributed neural network training. As all the software, hardware, and network variables in the whole system are considered by using different layers of classification and dynamic runtime system, we have been able to utilize the hierarchical hardware resources more efficiently in an uncertain network environment.

Work is still needed in the future in this subject. First, the algorithm which is used in the runtime system to control the objects and frequency of the synchronization can still be improved, as we can see from the experiment above. Even though we have improved the running time to convergence but there is still some room to improvement to reach the optimal value. More wisdom can be put into the effort to explore more efficient algorithm and also machine learning strategy can be brought into consideration.

Secondly, the experiments conducted so far may not fully represent exascale scenarios, despite considering scale factors. Future research could benefit from the inclusion of additional hardware resources, such as multiple clusters or multiple data centers, to achieve a more

comprehensive understanding of system behavior. Additionally, the current simplification of network conditions using big O notation might not capture all the intricacies. Therefore, future efforts should consider a more detailed examination of various network variables to enhance the accuracy and reliability of the entire system.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283, 2016.

[2] A. Agarwal, O. Chapelle, M. Dudík, and J. Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[3] T. Ben-Nun and T. Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[4] L.-C. Chen, Y. Zhu, G. Papandreou, F. Schroff, and H. Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[5] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[6] Z. Chen, J. Zhang, and D. Tao. Progressive lidar adaptation for road detection. *IEEE/CAA Journal of Automatica Sinica*, 6(3):693–702, 2019.

[7] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*, 2013.

[8] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[9] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. 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*, pages 1–16, 2016.

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[13] L. A. Gatys, A. S. Ecker, and M. Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

[14] T. Geng, L. Liu, S. Yin, M. Zhu, and S. Wei. Parallelization of computing-intensive tasks of the h. 264 high profile decoding algorithm on a reconfigurable multimedia system. *IEICE TRANSACTIONS on Information and Systems*, 93(12):3223–3231, 2010.

[15] T. Geng, S. Zuckerman, J. Monsalve, A. Goldman, S. Habib, J.-L. Gaudiot, and G. R. Gao. The importance of efficient fine-grain synchronization for many-core systems. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 203–217. Springer, 2016.

[16] J. Godino-Llorente, S. Shattuck-Hufnagel, J. Choi, L. Moro-Velázquez, and J. Gómez-García. Towards the identification of idiopathic parkinson's disease from the speech. new articulatory kinetic biomarkers. *PloS one*, 12(12):e0189583, 2017.

[17] H. Haim, S. Elmalem, R. Giryes, A. M. Bronstein, and E. Marom. Depth estimation from a single image using deep learned phase coded mask. *IEEE Transactions on Computational Imaging*, 4(3):298–310, 2018.

[18] A. Harlap, D. Narayanan, A. Phanishayee, V. Seshadri, N. Devanur, G. Ganger, and P. Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*, 2018.

[19] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[20] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems*, 32, 2019.

[21] Z. Jia, M. Zaharia, and A. Aiken. Beyond data and model parallelism for deep neural networks. *Proceedings of Machine Learning and Systems*, 1:1–13, 2019.

[22] R. Kadlec, M. Schmid, O. Bajgar, and J. Kleindienst. Text understanding with the attention sum reader network. *arXiv preprint arXiv:1603.01547*, 2016.

[23] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[24] D. Kwon, H. Kim, J. Kim, S. C. Suh, I. Kim, and K. J. Kim. A survey of deep learning-based network anomaly detection. *Cluster Computing*, 22:949–961, 2019.

[25] H. Li, A. Kadav, E. Kruus, and C. Ungureanu. Malt: distributed data-parallelism for existing ml applications. In *Proceedings of the tenth european conference on computer systems*, pages 1–16, 2015.

[26] R. Mayer and H.-A. Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)*, 53(1):1–37, 2020.

[27] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen, et al. Mllib: Machine learning in apache spark. *The journal of machine learning research*, 17(1):1235–1241, 2016.

[28] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, and J. Dean. A hierarchical model for device placement. In *International Conference on Learning Representations*, 2018.

[29] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

[30] E. Schwartz, R. Giryes, and A. M. Bronstein. Deepisp: Toward learning an end-to-end image processing pipeline. *IEEE Transactions on Image Processing*, 28(2):912–923, 2018.

[31] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[32] P. Villalobos, J. Sevilla, T. Besiroglu, L. Heim, A. Ho, and M. Hobbhahn. Machine learning model sizes and the parameter gap. *arXiv preprint arXiv:2207.02852*, 2022.

[33] H. Wang, H. Shi, K. Lin, C. Qin, L. Zhao, Y. Huang, and C. Liu. A high-precision arrhythmia classification method based on dual fully connected neural network. *Biomedical Signal Processing and Control*, 58:101874, 2020.

[34] H. Wang, Y. Wang, Z. Zhou, X. Ji, D. Gong, J. Zhou, Z. Li, and W. Liu. Cosface: Large margin cosine loss for deep face recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5265–5274, 2018.

[35] M. Wang, T. Xiao, J. Li, J. Zhang, C. Hong, and Z. Zhang. Minerva: A scalable and highly efficient training platform for deep learning. In *NIPS Workshop, Distributed Machine Learning and Matrix Computations*, page 51, 2014.

[36] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 84–97, 2016.

[37] E. P. Xing, Q. Ho, W. Dai, J.-K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1335–1344, 2015.

[38] S. Zeng, J. M. M. Diaz, and S. Raskar. Toward a high-performance emulation platformfor brain-inspired intelligent systemsexploring dataflow-based execution model and beyond. In *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 628–633. IEEE, 2019.

[39] C. Zhang, H. Tian, W. Wang, and F. Yan. Stay fresh: Speculative synchronization for fast distributed machine learning. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 99–109. IEEE, 2018.

[40] H. Zhang, L. Stafman, A. Or, and M. J. Freedman. Slaq: quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 390–404, 2017.