Efficient Asynchronous GCN Training on a GPU Cluster

Yi Zhang

A Thesis
in
The Department of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

March 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Yi Zhang

Entitled: Efficient asynchronous GCN training on a GPU cluster

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (M. Comp. Sc.)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________________ Chair

Dr. Sudhir Mudur

______________________________________ Examiner

Dr. Sudhir Mudur

______________________________________ Examiner

Dr. Tristan Glatard

______________________________________ Supervisor

Dr. Dhrubajyoti Goswami

Approved by ______________________________________

Dr. Sudhir Mudur, Chair of Department

____

Dr. Mourad Debbabi, Dean

Faculty of Engineering and Computer Science

Date ______________________________________

ABSTRACT

Efficient asynchronous GCN training on a GPU cluster

Yi Zhang

A common assumption in traditional synchronous parallel training of Graph Convolutional Networks (GCNs) using multiple GPUs is that load is perfectly balanced among all GPUs. However, this assumption does not hold in a real-world scenario where there can be imbalances in workloads among GPUs for various reasons. In a synchronous parallel implementation, a straggler in the system can limit the overall speed-up of parallel training. To address these issues, this research investigates approaches for asynchronous decentralized parallel training for GCNs. The techniques investigated are based on graph clustering and gossiping. The research specifically adapts the approach of Cluster-GCN, which uses graph partitioning for SGD-based training, and combines with a novel gossip algorithm specifically designed for a GPU cluster to periodically exchange gradients among randomly chosen partners. In addition, it incorporates a work-pool mechanism for load balancing among GPUs. The gossip algorithm is proven to be deadlock free. The implementation is done on a GPU cluster with 8 Tesla V100 GPUs per compute node, and PyTorch and DGL as the software platforms. Experiments are conducted for different benchmark datasets. The results demonstrate superior performance, at the compromise of minor accuracy loss in some runs, as compared to traditional synchronous training which uses all-reduce to synchronously accumulate parallel training results.

Acknowledgments

I would like to express my gratitude to my supervisor Dr. Dhrubajyoti Goswami for his guidance and encouragement.

I am thankful to my family for their support and love.

Table of Contents

List of Figures vii

List of Tables viii

Chapter 1 Introduction 1

1.1 Background 1

1.2 Problem Statement and Motivation 2

1.3 Challenges and contributions 5

1.4 Thesis Outline 6

Chapter 2 Literature Review 7

2.1 Graph Convolutional Networks (GCNs) 7

2.2 Mini batch Stochastic Gradient Descent (SGD) GCNs 8

2.2.1 Traditional Sampling Based GCN Training 9

2.2.2 Graph-partition based GCN Training 11

2.2.3 Subgraph Based GCN Training 12

2.3 Parallel SGD-based Deep Learning Training 12

2.3.1 Overview of Parallelism Strategies 13

2.3.2 Synchronous Stochastic Gradient Descent (SGD) 15

2.3.3 Asynchronous Stochastic Gradient Descent (SGD) using Gossiping 16

2.3.4 Periodic Gradient Averaging 20

2.4 GCN Training on GPUs 21

2.4.1 Existing Deep Learning Frameworks 21

2.4.2 Synchronization Using All-reduce 22

2.4.3 Implementation Using CPU/GPU Cluster 23

Chapter 3 Graph-partition based Synchronous Data Parallel Implementation 25

3.1 Baseline Implementation for GCN Training on a Single GPU 25

3.2 Synchronous GPU Parallel Implementations (AllReduce-GCN) 27

3.2.1 Synchronous GPU Parallel using All-reduce 27

3.2.2 Optimization with Periodic Gradient Averaging 31

3.2.3 Theoretical Discussion 33

3.3 Experimental Evaluations 33

3.3.1 Setup and Datasets 34
3.3.2 Experimental Results 36
3.4 Summary and Limitations of AllReduce-GCN 42
Chapter 4 Graph-partition based Asynchronous Data Parallel Implementation 44
4.1 Asynchronous Training based on Gossip 44
4.2 Asynchronous GPU Parallel Implementation Version 1 46
4.3 Asynchronous GPU Parallel Implementation Version 2 (GossipGCN) 49
4.3.1 GossipGCN on a GPU cluster 50
4.3.2 Work-pool 56
4.3.3 Optimization with Periodic Gradient Averaging 57
4.3.4 Theoretical Discussion 58
4.4 Experimental Evaluations 60
4.4.1 Implementation Details 60
4.4.2 Preliminary Experimental Results for Implementation Version 1 61
4.4.3 Experimental Results for GossipGCN 62
4.5 Summary 68
Chapter 5 Conclusion and Future Works 70
Reference 72

List of Figures

Figure 1.1 Gossip algorithm for distributed system and single compute node.........................4
Figure 2.1 Example of 2D convolution and graph convolution. (Figure courtesy of [20]).......8
Figure 2.2 Mini batch Stochastic Gradient Descent (SGD). (Figure courtesy of [13])............9
Figure 2.3 Sampling in GraphSAGE. (Figure courtesy of [23]) 10
Figure 2.4 Comparison of neighbourhood expansion. (Figure courtesy of [8]) 11
Figure 2.5 Architectures of parallelism strategies (Figure courtesy of [34])..........................13
Figure 2.6 Centralized Synchronous and asynchronous SGD. (Figure courtesy of [43]) 17
Figure 2.7 Centralized network and gossip-based decentralized network..............................18
Figure 2.8 Example of all-reduce call on four processors. (Figure courtesy of [12]).............22
Figure 3.1 Scheme of synchronous graph-partition based GCN using all-reduce..................29
Figure 3.2 Synchronous data parallel AllReduce-GCN. (Figure courtesy of [60]) 30
Figure 3.3 Barrier of AllReduce-GCN with periodic synchronization..................................32
Figure 3.4 Speed up of AllReduce-GCN for Reddit and Amazon dataset. 38
Figure 3.5 Speed up of AllReduce-GCN for Reddit Dataset...41
Figure 3.6 Speed up of AllReduce-GCN for Amazon Dataset...41
Figure 3.7 Fixed workload divide mechanism for AllReduce-GCN. 43
Figure 4.1 Deadlock in gossip averaging...45
Figure 4.2 An example run using 4 GPUs 53
Figure 4.3 Gradient averaging scheme for GossipGCN. 53
Figure 4.4 Gossip scheme with work-pool. 54
Figure 4.5 GossipGCN with periodic gradient averaging. 57
Figure 4.6 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit and Amazon dataset with averaging_interval = 1. 66
Figure 4.7 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit and Amazon dataset with averaging_interval = 20. 67
Figure 4.8 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit and Amazon dataset with averaging_interval = 100. 67

List of Tables

Table 3.1 Datasets for experimental evaluations 35
Table 3.2 Frameworks and libraries....................35
Table 3.3 Hyperparameters used for experiments 36
Table 3.4 Speed up of AllReduce-GCN for Reddit dataset. 38
Table 3.5 Speed up of AllReduce-GCN for Amazon dataset 38
Table 3.6 AllReduce-GCN for Reddit dataset (4 GPUs)....................39
Table 3.7 AllReduce-GCN for Reddit dataset (8 GPUs)....................39
Table 3.8 AllReduce-GCN for Amazon dataset (4 GPUs) 39
Table 3.9 AllReduce-GCN for Amazon dataset (8 GPUs) 39
Table 3.10 Speed up of AllReduce-GCN for Reddit Dataset 40
Table 3.11 Speed up of AllReduce-GCN for Amazon Dataset 40
Table 4.1 Asynchronous GPU parallel implementation version 1 for Reddit dataset (4 GPUs) 62
Table 4.2 GossipGCN for Reddit dataset (4 GPUs) 63
Table 4.3 GossipGCN for Reddit dataset (8 GPUs) 63
Table 4.4 GossipGCN for Amazon dataset (4 GPUs) 63
Table 4.5 GossipGCN for Amazon dataset (8 GPUs) 64
Table 4.6 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit dataset. 64
Table 4.7 Speed up comparison of AllReduce-GCN and GossipGCN for Amazon dataset. ..64
Table 4.8 Comparison of parallel training for Reddit Dataset (8 GPUs)....................65
Table 4.9 Comparison of parallel training for Amazon Dataset (8 GPUs)....................65

Chapter 1 Introduction

1.1 Background

Research on graph convolutional networks (GCNs) has increasingly gained popularity in recent years; this can mainly be attributed to powerful representational capacity of graphs. GCNs are widely used in many areas including social science [1, 2], chemical and biological research [3, 4], knowledge graphs [5] and many other disciplines.

GCNs evolve from convolutional neural networks (CNNs). Previously, researchers use graph embedding technologies to carry out deep learning training on graph-related data. However, there are defects in these methods, e.g., lack of generality for direct embedding and computational inefficiency caused by no parameters shared between nodes. The motivation of developing GCNs also roots in the natural features of graphs. Different from images and natural languages, graph nodes do not have a specific order, and an edge in a graph is not just the feature of nodes but also represents the information of dependency between two nodes. GCNs propagate on each node respectively, ignoring the input order of nodes. Moreover, the propagation is guided by the graph structure (edges) instead of using it as part of features [6].

The goal of GCN is to solve various graph-related tasks by using neural network models. GCNs can be used for different tasks, including node classification, link prediction and graph classification. GCNs develop a state embedding, which holds information of graph nodes and the structure of the graph. This state embedding can be used to generate an output such as the node label. The outline of GCN was first defined in [7], and the original work extends recursive neural networks (RNN). Since then, a lot of different graph neural network models have been proposed, but there doesn't exist a systematic categorization for GCNs until recently.

GCNs play an important role in deep learning research, however one common problem with any variant of GCN training is due to neighbourhood expansion, i.e., in computing the loss on a node at current layer, it recursively requires the neighbouring nodes' information at previous layers, which leads to an exponential increase in time complexity due to GCN depth. This problem is one of the causes for shallow structure and scalability limitation for GCNs and is known as the Neighborhood Expansion Problem (NEP). Traditional CNNs can have up to hundreds of layers, while GCNs usually have no more than 3 layers. Moreover, increasing memory consumption in the recursive steps restricts the graph size that can be fitted in the training.

Cluster-GCN [8] and GraphSAINT [9] propose two different ways to overcome NEP by performing training on smaller subgraphs instead of using the original large graph. Cluster-GCN uses clustering methods to partition the original graph into subgraphs and prepare mini batches from the subgraphs. GraphSAINT constructs mini batches by sampling the input graph but builds a complete GCN at each iteration with the sampled data. Both methods lead to a heuristic model as final output and have the advantage of restricting neighbourhood expansion to a relatively smaller range in subgraphs. In our research, we use a similar technique based on subgraph training.

With the advancement of hardware and the increasing size of datasets, there is a high demand for parallel deep learning training. Large input data for deep learning models consume longer time for the training process, which encourages exploring parallel strategies to improve the speed up. Previous research has explored parallel training for GCNs on CPUs and distributed systems, e.g., GraphLab [10] and PowerGraph [11]. In recent years, parallel GCN training on GPUs has gained popularity with the wide employment of deep learning frameworks like PyTorch [12] whose support on GPUs has matured. This has opened the possibilities of exploring GPU computational power in GCN training.

Generally, there are two well-known parallelization strategies, namely data parallelism (DP) and model parallelism (MP). As presented in [13], a new trend appears recently combining DP and MP. Although model parallelism and hybrid parallelism are useful in specific scenarios, considering the simplicity and generality, data parallelism is the dominant approach in parallel training for GCNs. Despite the advantages of data parallelism, one of the major bottlenecks for this methodology is the gradient averaging in each iteration. Workers need to communicate local gradients with other workers, and this communication overhead degrades training speed up. In recent years, many algorithms have been proposed to moderate the drawbacks caused by gradient synchronization. In this research, we explore the opportunity of improving GCN training speed up by using asynchronous gradient averaging method based on the idea of gossiping.

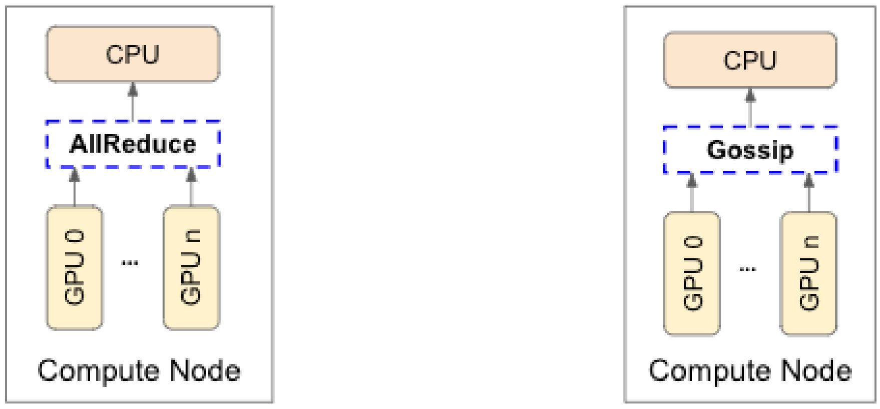
1.2 Problem Statement and Motivation

The performance of traditional GPU synchronous data parallel training for GCNs depends on the speed and workload of each GPU. A common assumption is that GPUs of the same make and model would process their mini batches at the same pace. However, experiments show that this assumption is not valid in a real-world scenario, where the cluster nodes may not be dedicated to a single job; moreover, the training workload sizes on GPUs may differ. In a synchronous parallel training, GPUs may calculate gradients at different paces, which causes faster workers to sit idle when waiting for a straggler. Section 3.3 illustrates this scenario with experimentation.

In traditional sampling-based GCN models, the sampling (batch-preparation) time and "real-training" time (gradient calculation and model updating) are usually measured together as the model training time. But in graph-partition based GCN such as Cluster-GCN, the batch-preparation is done independently before the training starts. So, when applying the synchronous gradient averaging methods to this kind of GCN models, the delays caused by stragglers become obvious in the training phase. The motivation of our research is to overcome the drawback caused by the synchronization of gradients.

To overcome the performance bottleneck caused by the synchronization delay due to idling, in this research we investigate an asynchronous parallel training methodology of GCNs based on the idea of gossiping (GossipGCN). Although gossiping-based asynchronous parallel training has been explored for CNNs in distributed systems, to the best of our knowledge there is limited research on asynchronous parallel training of GCNs on a GPU cluster.

Previous research has investigated gossip algorithm for CNN training using GPUs in a distributed system. Such a system consists of multiple compute nodes, where each compute node consists of one CPU connected with one or more GPUs, and multiple connected CPUs form a distributed cluster. Usually the gradient calculation and weight updates are done on GPUs inside a compute node while the communication for averaging gradients using gossip runs among CPUs [14-16]. Averaging of gradients among GPUs within a compute node is usually achieved by using all-reduce (Figure 1(a)) or using the CPU as a central server. In a centralized system with a parameter server, the central server may cause communication bottleneck. For a decentralized system as in Figure 1.1 (a), the collective communication (all-reduce) among GPUs can be achieved using NVIDIA Collective Communications Library (NCCL) [17], which provides fast communication backend among GPUs. As discussed before, there are performance bottlenecks with this approach as well.



In a distributed system, gradients are averaged among GPUs using AllReduce method within each node, then gossip is used to synchronize gradients among multiple compute nodes.

In our research, we explore the possibility to use gossip algorithm within a single compute node to average gradients among GPUs.

Figure 1.1 Gossip algorithm for distributed system and single compute node.

As a major difference, in our work, we investigate the gradient averaging among GPUs in a compute node using gossip (Figure 1.1 (b)). To the best of our knowledge, this is the first such attempt to use gossip in asynchronous training of CNN or GCN inside a compute node. This is further elaborated in the following.

In addition, previous experiments of asynchronous gradient averaging methods are done for CNN related problems such as image classification. As far as we know, there is very limited research about using asynchronous algorithms for GCNs. Although there are similarities between CNNs and GCNs, the model construction, information propagation and dataset performance are different. And the benchmark results gathered in previous research cannot be used for GCN related problems. So, we propose to have an initial investigation on using gossip gradient averaging for GCN training on a GPU cluster.

Our research adapts the algorithm of Cluster-GCN [8] and focuses on the model training phase. We design and implement an asynchronous decentralized data parallel training method for graph convolutional networks (GCNs) with an adjustable averaging interval and compare its performance with the synchronous counterpart. Since PyTorch is the new trend to carry out graph related deep learning, the proposed algorithm is implemented using PyTorch.

1.3 Challenges and contributions

In order to implement gossip algorithm for gradient averaging among GPUs, the major challenge is to ensure deadlock avoidance. The deadlock happens when the dependencies of workers end up in a loop. Previous studies propose different methods to avoid deadlocks during gossiping, however, they are not suitable for asynchronous GCN training on GPUs. To implement an efficient deadlock avoidance algorithm for GCN training on GPUs, we propose a new method to use shared variables to control gossip-neighbor information. The efficiency of this method is achieved by keeping minimum message passing among workers and dynamically choosing available workers. In addition, by mapping each GPU with a process, it is convenient to use quick get and put methods for light weighted data such as process index.

Moreover, to improve the speed up and reduce time wasted on waiting idly, we implemented a work-pool mechanism to make the best use of the computing capacity of each worker. In the synchronous AllReduce-GCN method, each GPU has to train the same number of mini batches since they need to average gradients in every iteration. In our proposed method, we allow fast workers to request more mini batches whenever they are free, thus the calculation capacity of each GPU is fully used. We also incorporate the periodic synchronization method with the work-pool mechanism, and the synchronization interval can be determined empirically.

We present an asynchronous and decentralized data parallel algorithm (GossipGCN) for graph neural network training. Experiments are carried out on node classification tasks. The following are the highlights of the approach:

- Partitioning of the original input graph into subgraphs based on random clustering and doing parallel training on the subgraphs.
- Design and implementation of a gossiping-based algorithm using PyTorch which averages gradients among randomly chosen partner GPUs at set intervals.
- The gossip algorithm is proven to be deadlock-free.
- Partner selection is more random than the previous gossip algorithms on GPUs known to us.
- Enhanced efficiency by adopting a work-pool based strategy where workers (GPUs) are assigned works dynamically rather than a static work assignment.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 presents research background and related works. Chapter 3 elaborates the implementation of graph-partition based synchronous data parallel GCN training and discusses its limitations. Experiments are done to evaluate the effects of periodic gradient averaging on synchronous approach. Chapter 4 discusses the design and implementation of the proposed algorithm and related proofs. Experimental results with various benchmark datasets are also included in this chapter. Finally, Chapter 5 concludes this research by summarizing our work, followed by a discussion on future works.

Chapter 2 Literature Review

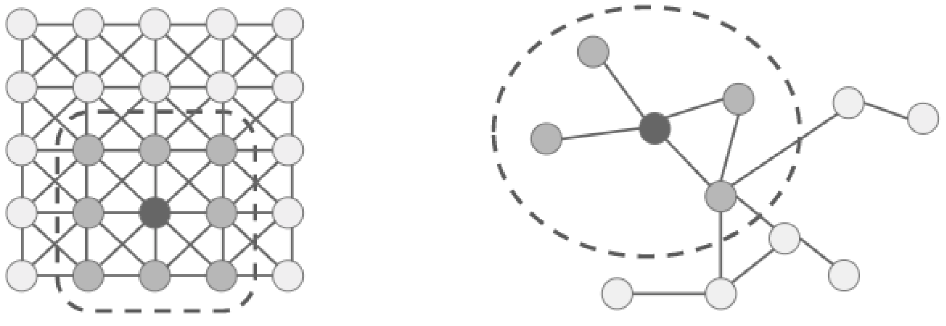
Graph convolutional networks (GCNs) have become a popular tool for deep learning tasks on graph data. They are widely applied to different domains including social science [1, 2], chemical and biology research [3, 4], knowledge graphs [5] and traffic networks [18, 19]. GCNs belong to an important branch of graph neural networks (GNNs). GNNs are evolving fast in recent years and different variants are proposed, such as graph convolutional networks (GCN), graph attention network (GAT) and jump knowledge network (JK-Networks). [20] conducts a comprehensive survey on GNNs and proposes a new taxonomy for categorization of related works. [21] provides a generalized GNN benchmarking framework, which facilitates evaluation of different GNN architectures. Among the different versions of GNNs, graph convolutional networks (GCNs) set the foundation in early works and have a long-lasting influence on deep learning research. With increasing size of input graphs and the advancement of hardware and parallel methodologies, parallel training of GCNs has gained a lot of attention and different parallel strategies are proposed.

In this chapter, we first provide a brief background of GCNs and its relationship with convolutional neural networks (CNNs). Then we discuss mini batch Stochastic Gradient Descent (SGD) GCNs and its variants. Subsequently, we provide a review of different schemes for parallel GCN training and various optimization methods. Finally, we conclude this chapter with a discussion of GCN training on GPUs.

2.1 Graph Convolutional Networks (GCNs)

Encouraged by the popularity of convolutional neural networks (CNNs) and recurrent neural networks (RNNs), a large number of studies are done for deep learning research on graph data, and the concept of graph convolutional networks (GCNs) appears. Traditional neural networks like CNNs organize the node features in a specific order, which is actually not suitable for graph related data. In a graph, there doesn't exist any natural order for the nodes, and GCNs use message passing between nodes to capture the dependency of graphs.

To have a more straightforward understanding, an example of relations between CNNs and GCNs is illustrated in Figure 2.1. In CNNs for image processing problems, it usually uses a convolution kernel and pooling layers to transform information. Graph dependency information (represented by edges) is considered as a feature of nodes. While in GCNs, information is propagated on each node respectively through message passing, and the state of nodes is updated by a weighted sum of the neighbor nodes' states. The propagation ignores the input order of nodes and is guided by the graph structure (edges in the graph).



In 2D convolution, a filter is used to gather information of each pixel and its neighbors.

In graph convolution, information is gathered along edges using propagation.

Figure 2.1 Example of 2D convolution and graph convolution. (Figure courtesy of [20])

Using a fixed number of layers with different weights, graph convolutional networks models the dependency of graph nodes. In each layer, it acquires the embeddings by gathering information from workers. The first complete definition of GCN is presented in [22]. The authors propose a model derived from convolutional neural networks to solve graph-related deep learning problems. The proposed model is scalable. At the same time, it encodes features of nodes and graph structure. This work is critical for later research on graph convolutional networks and promotes the widespread adoption of GCNs in graph related deep learning problems. Different variants of GCNs include GraphSAGE [23], FastGCN [24] and graph convolutional networks with variance reduction [25].

2.2 Mini batch Stochastic Gradient Descent (SGD) GCNs

The pioneer work of GCN training [22] uses full-batch gradient descent, where gradients are calculated with the complete input data and weights are updated after each epoch. This method is not scalable because of slow convergence and increasing memory requirement for large input data. To cope with the scalability problem of GCN training, [23] propose a mini batch stochastic gradient descent (SGD) algorithm for large-scale graphs, where weights are updated after each iteration based on gradients generated from training a mini batch. A mini batch is a subset of training examples and is smaller than the original input graph. This method requires less memory compared with full-batch gradient descent and has a better converge rate since multiple updates are conducted in one epoch. An illustration of mini batch SGD is shown in Figure 2.2.

In mini batch SGD, computing loss on a single node is dependent on the embeddings of its neighbors at the previous layer, and this process continues recursively based on the GCN depth, thus it leads to time consumption grows exponentially to GCN layer numbers. The problem caused by such dependency is known as neighbourhood expansion problem. Previous studies propose various solutions to reduce side effects of such a problem, a typical solution is to use sampling methods to prune the neighbors of a node to reduce expended neighbors [23, 24]. Sampling methods limit the size of neighbor samples for each node; however, the overhead still grows quickly with the GCN depth. Two innovative methods [8, 9] are proposed since 2019 and offer new perspectives to deal with the neighbourhood expansion problem using graph partition methods and subgraph based GCN training.

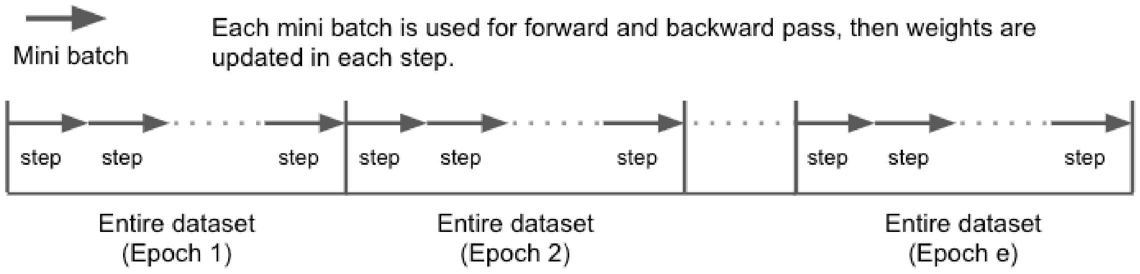


Figure 2.2 Mini batch Stochastic Gradient Descent (SGD). (Figure courtesy of [13])

2.2.1 Traditional Sampling Based GCN Training

The essential idea of sampling is to have an accurate estimation of GCN embeddings by reducing dependent neighbor nodes. There are many different sampling methods that can be used for mini batch SGD training of GCN models, and some of them are supported in flatforms and libraries such as PyTorch [26] and DGL [27]. As mentioned in a survey [28], there are mainly two types of sampling methods: neighborhood sampling and layer-wise sampling. An example of the sampling method is shown in Figure 2.3.

Neighborhood sampling methods samples the neighbors for each node during training. One of the well-known neighborhood sampling methods is presented in GraphSAGE [23]. The author suggests to sample a fixed-size set of neighbors for the calculations. Using this method can guarantee to have a predictable space and time complexity for each iteration. This work sets the foundation for lots of later research on neighborhood sampling, and different variants are proposed to optimize GraphSAGE. [29] presents an advancing data-driven sampling method to address the high variance in training, and it achieves a better accuracy than the original version of GraphSAGE.

Another variant of neighborhood sampling is illustrated in PinSage [30] to use highly efficient random walks to estimate the graph embeddings for GCNs. In addition, PinSage proposes an efficient MapReduce model inference algorithm for constructing embeddings. According to their experiments, this method works well on large-scale graphs with billions of nodes and edges.

Even though the neighborhood sampling has great achievement in mitigating neighbourhood expansion, the number of dependent nodes still grows exponentially to GCN depth, which causes time and memory consumptions increase rapidly. To overcome the issue of over expansion, FastGCN [24] proposed a layer-wise sampling method, where a layer is considered integrated and restrain the number of sampled nodes per layer. The proposed method also incorporates importance sampling to reduce approximation variance. [31] illustrates another adaptive sampling method in which the sampling is done in a top-down manner. This method is proven to be effective on classification accuracy and has faster convergence rate.

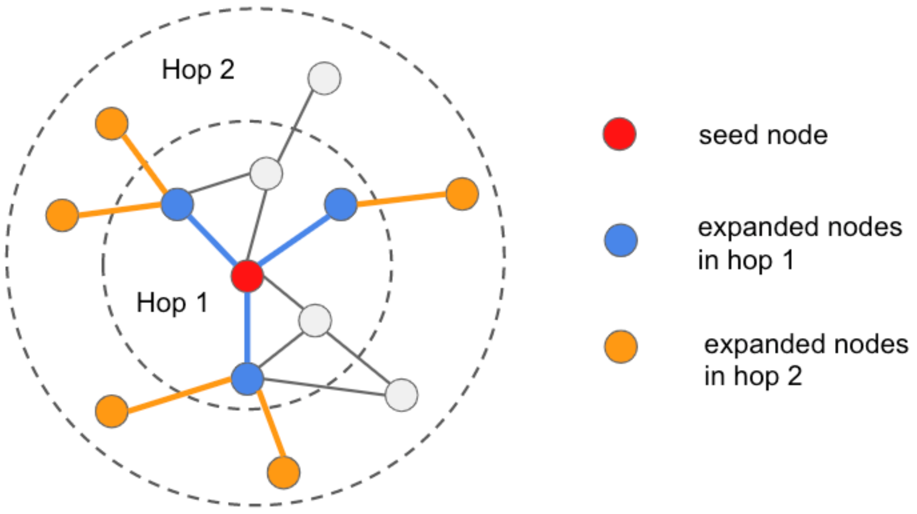


Figure 2.3 Sampling in GraphSAGE. (Figure courtesy of [23])

2.2.2 Graph-partition based GCN Training

In GCN training based on traditional sampling methods, although efforts are made to cope with the neighbourhood expansion Problem, a large portion of training time is still taken by gathering neighbor nodes' information recursively from previous layers. Cluster-GCN [8] proposes a different methodology for solving this issue. At each iteration, the algorithm conducts training on a subgraph instead of the entire GCN layers as in traditional sampling methods. The subgraphs are identified by a graph clustering algorithm, such as Metis or random partition. As illustrated in Figure 2.4, Cluster-GCN constrains neighbourhood expansion within the subgraph, and the number of expanded neighbor nodes are less than using the traditional sampling methods, thus avoiding expensive increase of time and memory consumptions.

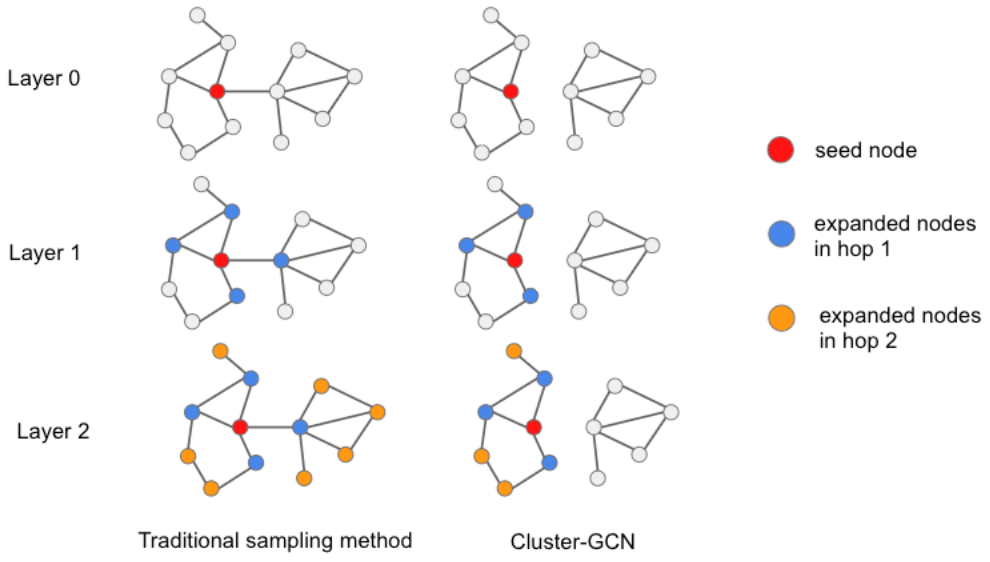


Figure 2.4 Comparison of neighbourhood expansion. (Figure courtesy of [8])

Sampling from a subgraph leads to a loss of graph information, thus it causes lower accuracy. Moreover, although using clustering methods (e.g. Metis) can result in minimum edges cut in graph partition, it introduces the issue that similar nodes tend to be partitioned in the same group. Since subgraphs may be an inaccurate representation for the original graph, it can lead to biased estimation for GCN training. To compensate for the missing links and increased variance brought by biased clustering, Cluster-GCN introduces a method that groups the partitioned subgraphs randomly and recovers the missing links within the group.

The authors carry out experiments using median and large size data to compare performance of Cluster-GCN and other state-of-the-art sampling methods, such as GraphSAGE and FastGCN. Results show that the proposed method can achieve equivalent test accuracy compared with previous methods and enjoys a much faster training speed. In addition, Cluster-GCN improves memory efficiency impressively and enables training on deeper GCN. Their examination on a 5-layer Cluster-GCN improves the prediction accuracy on the PPI datasets to 99.36 (F1-score).

In our research, we adapt the approach of Cluster-GCN for the ease of graph partitioning on GPUs, but unlike the original work we use a random clustering instead of Metis [32] to create the subgraphs. In our experiments, random clustering is found to provide better performance than Metis and could lead to the desired accuracy with properly chosen model and hyperparameters. One explanation is that the result of the training is a heuristic model and with large enough subgraph size the needed information can be maintained.

2.2.3 Subgraph Based GCN Training

GraphSAINT [9] uses a similar idea of restricting neighbourhood expansion as in Cluster-GCN by using subgraphs. It uses the sampling method to get subgraphs and builds the GCN on these subgraphs. In addition, the authors propose a normalization technique to reduce bias and exploit the possibility of combining the algorithm with different GCN variants.

In the research of GraphSAINT, the author states that their method is faster than Cluster-GCN and has better accuracy. But in the paper, they only conduct experiments using original codes of Cluster-GCN, which is implemented with Metis method and original GCN model, thus it consumes a long time to prepare the mini batches and leads to lower accuracy than GraphSAINT. We reproduced the comparison of the two methods using the same aggregator (GraphSage). The code for GraphSAINT is taken from the authors' GitHub post [9], and for Cluster-GCN, we use the example code in DGL GitHub library [33]. Instead of using Metis for graph clustering as suggested in the original paper [8], we use a random clustering method for Cluster-GCN. It turns out that the revised version of Cluster-GCN has a similar performance as GraphSAINT in both time consumption and accuracy. Besides, Cluster-GCN is easier to implement compared with GraphSAINT.

2.3 Parallel SGD-based Deep Learning Training

Parallel training for GCN models has gained popularity recently. Because of the quickly increasing number of neighbor nodes in each layer, usually it is difficult for GCNs to train models with deep depth. In addition, increasing data size for modern deep learning tasks consumes longer time to train the models. Nowadays, with the development of hardware and parallel programming technologies, GCNs is able to accomplish various training tasks. In this section, we discuss the commonly used parallelism strategies and the different synchronization methods for gradient averaging in deep learning training. Optimization methods such as local stochastic gradient descent and periodic averaging are also discussed in this section.

2.3.1 Overview of Parallelism Strategies

As mentioned above, the two major categories for parallel SGD-based deep learning training are data parallelism (DP) and model parallelism (MP). Figure 2.5 illustrates the architectures for the two parallel strategies. Recent research [13] also proposes a combination for DP and MP. Among these choices, data parallelism is the prevailing approach because of its simplicity and generality.

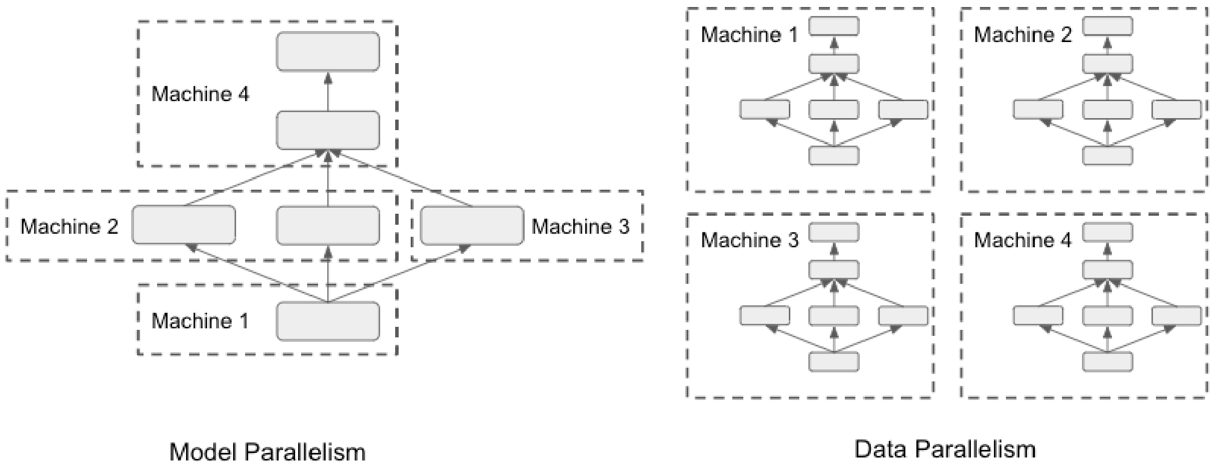


Figure 2.5 Architectures of parallelism strategies (Figure courtesy of [34])

GCN models are suitable for synchronous data parallel, since there is no order for the nodes, and the propagation is done on each node respectively. In data parallelism, each worker has a replica of a deep learning model. The local models are trained in parallel with independent subsets of input data. Multiple mini batches can be trained simultaneously, thus it reduces time consumption for passing the whole input set. The workers use synchronization methods to average gradients and then apply the same gradient to update weights. The batch for a step in each worker is called a mini batch, and a collection of mini batches from all workers is called a global batch. So, the number of parallel workers affect the global batch size, which has an influence on the final accuracy achieved for the model. To get a desired accuracy, it may require more iterations for data parallel training compared with baseline training on the same datasets. Additionally, the increasing number of workers brings more overheads in synchronization and communication processes. Even though data parallelism has these potential drawbacks, it is still the most popular parallelization strategy, because it is relatively easy to implement and broadly supported in major deep learning frameworks, such as PyTorch [12] and TensorFlow [35].

Model parallelism (MP) is used for deep learning models that are too large to be fit in a single device. In MP, the model data flow graph (DFG) is split in parallel on different workers, and all devices work together for passing a mini batch in training. Since the model is divided across multiple workers, the forward and backward propagations require communication between workers in a sequential fashion. This is the reason why MP is also considered as "Model Serialization", because it actually uses a serial approach for gradients calculation. Since each training step takes less time than using a single device, speed up can be achieved by using this approach. However, the scalability in MP is limited by GCN model's algorithm and implementation. It is also difficult to maximize speed up of MP since it depends on model DFG and system hardware. If the parallel overhead is too large, it may overshadow the profit of using MP.

For a deep learning model which is sequential in nature and doesn't have parallel branches, another strategy to parallel is using pipelining. In this approach, the layers of the model are grouped and assigned to independent devices. In one training step, a mini batch is divided into several micro-batches. Each worker processes one micro-batch simultaneously and follows the sequence. In a certain way, this approach can also be considered as an implementation instance of MP. Research [36] has been done to use graph computation optimizations and pipelining methods to optimize model parallelism. It proposes a framework to combine the graph and dataflow models and achieves promising performance on both small and large real-world graphs.

A new hybrid approach is presented in [13] to combine data parallelism and model parallelism to push forward the speed up limitation of using DP or MP alone. The authors propose to have multiple devices for a single worker. Inside each worker, it applies model parallelism with its devices to accelerate each training step. Then data parallelism is applied among the workers. If a system has a large number of devices and using data parallelism alone can't take full advantage of the computation resources, then using the hybrid parallelism can push the limit of speed up further. Their experiments show that if speed up gain from MP can overcome increased overhead and efficiency losses, using this hybrid implementation can improve training speed up.

In reality, there is no single solution for the choice of parallelism approaches because of model properties and system configurations. Because of the parallel nature of graph nodes and propagation process, DP is widely adopted and the implementation is easier with help of deep learning platforms and libraries, such as PyG [26] and DGL [27]. Although data parallelism has lots of advantages, one of the main bottlenecks for DP is the gradient synchronization. Two major types of gradient synchronization methodologies are discussed below, specifically synchronous and asynchronous data parallelism.

2.3.2 Synchronous Stochastic Gradient Descent (SGD)

According to [37], current gradient synchronization algorithms can be grouped into four categories: (1) communication synchronization (synchronous, asynchronous, etc.); (2) system architectures (all-reduce, gossip, etc.); (3) compression techniques; (4) parallelism of communication and computing. Different synchronization patterns can integrate with different system architectures. For example, synchronous communication can work with the all-reduce method, which is a very common gradient synchronization approach in GCN training.

Synchronous SGD has been studied for years and is a common way to implement data parallelism for deep learning training. The convergence of synchronous SGD for deep learning problems are discussed in [38, 39]. For synchronous SGD, each worker fetches a mini batch and calculates local gradients, then local gradients get averaged and the model is updated. Depending on the system architecture, there are centralized and decentralized variants for synchronous data parallel implementations.

In a centralized implementation, the model is stored in a parameter server, usually it can be the CPU memory. In each iteration, each worker (e.g. GPU) fetches the model saved in the parameter server and gets a mini batch to compute gradients. After finishing computing, workers send the stochastic gradients back to the parameter server, and the gradients are averaged on the parameter server when all results are returned. The model is updated with the averaged gradients, then the next iteration starts by sending out the updated model and new mini batches to workers.

Another variant of synchronous SDG is proposed in [40] to maximize the time spent on useful computations. The algorithm is called Stale Synchronous Parallel (SSP). It allows parallel workers to read stale values from a local cache and continue training using the old model. This approach reduces time that workers spend on waiting idly for the values from a central storage over the network, thus improving the ratio of time for useful computations. However, there is an iteration gap when parallel workers get a different number of mini batches locally. If this iteration gap is too large, the fastest worker needs to pause and wait for the slowest one. The authors provide a proof for the correctness of this algorithm and conduct experiments on various deep learning problems. The paper states that SSP achieves faster converge speed compared with fully synchronous systems at that time.

The problem of centralized synchronous SGD is that the communication through the parameter server can be the bottleneck. To overcome the communication bottleneck, decentralized synchronous SGD is presented in [41, 42]. For decentralized synchronous SGD, each worker keeps a local copy of the model and fetches a mini batch to calculate stochastic gradient locally. In each iteration, the gradients are averaged through a collective communication among workers, which is usually done using all-reduce. Then the gradients are identical on each worker, and the local model is updated using the averaged gradients. Although there is no need for a centralized parameter server, this approach still falls in a model-centralized topology, since all local models are identical after synchronization at each step, which is equivalent to having a global model.

The decentralized synchronous SGD using the all-reduce method (AllReduce-SGD) is widely accepted and promoted. In PyTorch [26], the contributors optimize the all-reduce call by organizing parameter gradients into buckets and parallelizing computing and communication. Even with all these efforts, there is an unavoidable defect of synchronous SGD. If there exists a straggler (slow worker) in the system, it will affect the overall training speed extremely. Since synchronization needs to be performed at each iteration, the fast workers have to wait in idle for the slow ones. Besides, frequent synchronization requires large bandwidth for communication among workers.

2.3.3 Asynchronous Stochastic Gradient Descent (SGD) using Gossiping

In synchronous SGD, all parallel workers need to communicate with other workers to average stochastic gradients in each iteration, which leads to high communication cost. Besides, since there is a barrier at each step to synchronize, the fast workers have to stay idle and wait for the slow ones. All these add to the time consumption and bandwidth burden of synchronous SGD. To overcome these disadvantages, research has been done to minimize the overhead caused by gradient synchronization, and asynchronous stochastic gradient descent (asynchronous SGD) is proposed recently. Basically, asynchronous SGD breaks the synchronization in each iteration and reduces the idling and communication time significantly in the training process. Figure 2.6 illustrates the schemes of synchronous and asynchronous SGD.

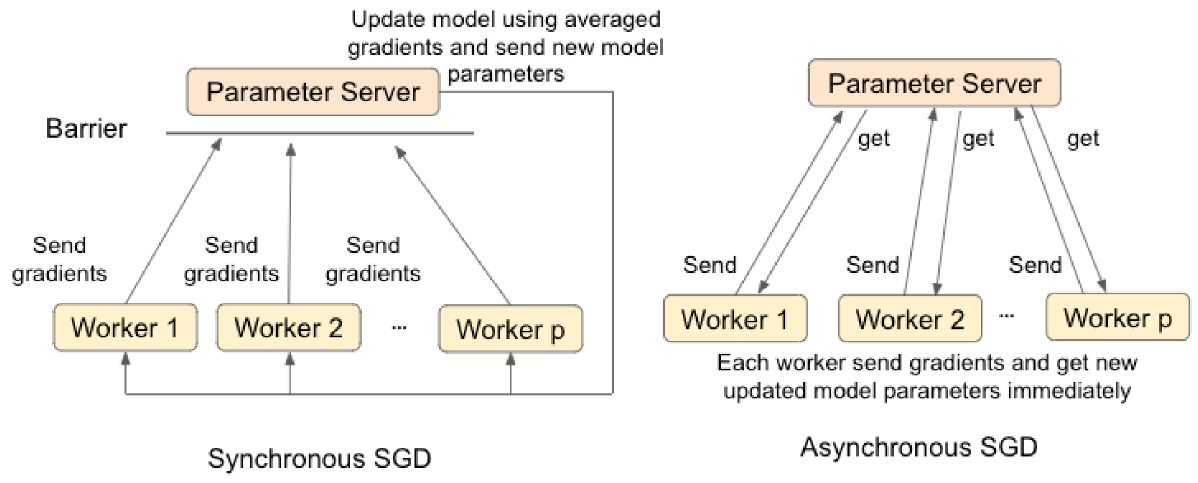


Figure 2.6 Centralized Synchronous and asynchronous SGD. (Figure courtesy of [43])

Similar to the synchronous version, asynchronous SGD has many different variants, including centralized and decentralized schemes. Centralized asynchronous SGD uses a parameter server to store the global model and manage the gradient averaging and parameter updating, while in decentralized versions, workers communicate with each other in a decentralized fashion such as gossiping. Thus, decentralized asynchronous SGD gets rid of the communication bottleneck at parameter server.

In centralized asynchronous SGD, a model is stored initially on the parameter server. Parallel workers obtain current model parameters and a mini batch to calculate gradients locally. Once the calculation work is done in a worker, it sends the gradients back to the parameter server. The global model is updated asynchronously, and each worker does not need to wait for its peers. The worker gets a new set of parameters from the parameter server immediately after finishing previous work and fetches a mini batch to continue the calculation of gradients.

Centralized asynchronous SGD is widely used for various deep learning problems. However, it still suffers from the communication bottleneck and slow convergence caused by a centered parameter server. In addition, centralized systems are vulnerable to potential central point failure, which will cause the whole system to shut down. To eliminate the communication bottleneck and central point failure issues, decentralized asynchronous algorithms are widely adopted in recent studies of various deep learning problems. These kinds of algorithms are more tolerant to slow workers and worker failures. Many implementations of decentralized asynchronous SGD use gossip algorithm, where workers can choose a random neighbor to average gradients [15, 16]. Figure 2.7 shows a comparison of centralized network and gossip-based decentralized network.

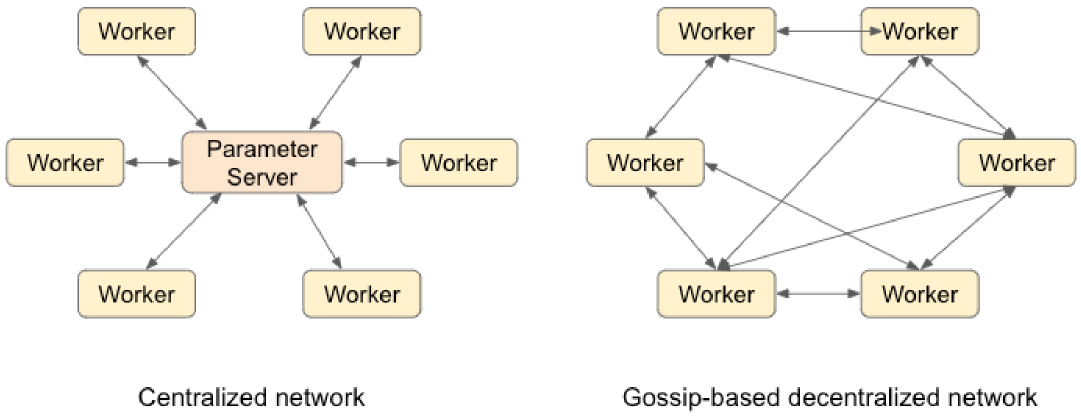


Figure 2.7 Centralized network and gossip-based decentralized network

[44] proposes a decentralized non-gradient-based algorithm for solving optimization problems. They develop a distributed asynchronous iterative algorithm with gossiping methods for achieving optimization over undirected networks. [45] presents theoretical analysis and proves decentralized SGD algorithms can outperform centralized counterparts since less communication cost is required on the busiest node. Their experiments show that in low bandwidth or high latency systems, decentralized SGD outperforms centralized algorithms up to one order of magnitude.

Asynchronous parallel training based on gossiping has been investigated in previous research to improve training performance. Gossip algorithm [46] is initially used for consensus problems, e.g., to compute the mean of data distributed in different computing nodes. In gossip, a node (computer) randomly chooses a partner to exchange information, and after a period of time, it is guaranteed to achieve robust information exchange among all nodes. The correctness and usage of gossip algorithm for distributed data aggregation is discussed in prior studies [46-49]. In deep learning training, using gossip can break the synchronization barrier of all-reduce across iterations by requiring to synchronize only between pair(s) of nodes in point-to-point data communication and hence reduce synchronization overhead of all-reduce in the presence of stragglers.

[50] presents a method using request-based methods to apply the gossip algorithm on the distributed averaging problem. The authors propose to send requests to workers before real gossip happens, and the gossip only occurs when a worker accepts the request. This method guarantees that deadlocks are avoided, however the requests sent among processes add to the communication overhead.

An asynchronous decentralized SGD algorithm is proposed in [15]. The authors present AD-PSGD algorithm that performs well in a heterogeneous environment and enjoys an optimal converge rate. According to their theoretical analysis and experiments, the algorithm achieves linear speed up and a similar epoch-wise convergence rate compares with the synchronous all-reduce counterpart. The proposed method adopts the traditional sampling methods for getting mini batches and uses a gossiping-style algorithm for averaging stochastic gradients. To eliminate deadlocks, the authors suggest dividing the workers into two groups, explicitly active set and passive set. Active set is responsible for sending the gradient averaging request to a random worker in the passive set, then the passive neighbor will return its local gradients, and the two workers update their local models with the same averaged gradients. This work provides an efficient asynchronous SGD algorithm using gossiping and has an important influence on our research.

Stochastic Gradient Push (SGP) is presented in [16] incorporates The PUSHSUM gossip algorithm with stochastic gradient descent for solving deep learning problems. Previous synchronous and asynchronous algorithms use different methods to calculate an exact inter-node average gradient, while SGP propose to compute approximate averages using PUSHSUM method. The author proves that with a properly chosen step-size, their method has a similar convergence as the SGD algorithm. Experiments are done for image classification and neural machine translation tasks, and the results show that SGP is robust in systems with stragglers and overall speed up for deep learning training is improved.

The advantages of asynchronous SGD are shown in the above discussions. To summarize, it is more tolerant for heterogeneous environments and can reduce the effect of stragglers, and less communication cost relieves burden on the network bandwidth. In spite of these benefits, this methodology has a potential disadvantage caused by asynchronous model updating. For example, a worker A takes mini batch i and its parallel neighbor B gets mini batch i+1. When worker A finishes calculating gradients and tries to get the next mini batch i+2 and updated parameters, worker B may not finish calculations and the parameters get by A are not updated based on mini batch i+1. This means that the next iteration carried out on A is based on outdated parameters and the gradients may have variance.

Research has been done to address the issue caused by update delays and proves the correctness of asynchronous SGD. Theoretical analyses of asynchronous SGD are presented in [51, 52]. The authors theoretically prove that despite asynchronous delays, the convergence is achievable with linear speed up.

2.3.4 Periodic Gradient Averaging

In synchronous and asynchronous parallel SGD solutions, previous research shows that a linear speed up is achievable in theory, however, the scalability of such speed up is limited due to the communication overhead caused by synchronization. To reduce the communication cost, another type of optimization is proposed that reducing synchronization frequency can lower the communication overhead significantly. Research [53, 54] suggests performing gradient averaging periodically among the workers. This kind of approach is known as local SGD or periodic averaging. It has been discussed theoretically in recent years and shows promising results practically.

In AllReduce-SGD, synchronization happens in each iteration, it has high statistical efficiency but requires expensive communication cost. While in local SGD, synchronization is performed at a certain time interval. The extreme situation is that only one-time synchronization happens at the end, which is also called one-shot averaging. It requires very little communication in one-shot averaging, but the gradients are averaged only once. So, there is a trade-off for training speed and accuracy, and local SGD tries to find the balance point to maximize the performance. A theoretical proof of synchronous and asynchronous local SGD convergence is given in the paper [53].

Some early studies related to interval synchronization include the work of [55]. The authors study parallel deep learning problems under communication constraints and proposed Elastic Averaging SGD method (EASGD) to reduce the number of communications between parallel workers and the central machine. The algorithm enables the master worker to update the model when parallel workers finish local updates after a certain communication period. The authors present different variants of the algorithms, including both synchronous and asynchronous versions. Their experiments verify the communication efficiency of the proposed algorithm.

[54] carries out study on periodic averaging and provides theoretical analysis on deep learning problems. The authors also present a scheme for deciding synchronization frequency and the conditions that affect speed up performance. They conduct experiments to show that

with properly chosen communication frequency, periodic averaging can achieve close to linear speed up.

In addition to experimental proof of the efficiency on model averaging, [56] elaborates on the theoretical exploration for the methodology. Their research provides a complete and rigorous theoretic guarantee of convergence for model averaging on deep learning problems and gives guidelines on how often the gradient averaging should be done during the training to achieve linear speed up.

2.4 GCN Training on GPUs

2.4.1 Existing Deep Learning Frameworks

The community of deep learning research has grown rapidly in recent years. Compared with pioneer researchers in this field, nowadays implementations of GCNs and other neural networks are becoming more convenient with the support of various open-source software libraries and deep learning frameworks. The well-known frameworks include TensorFlow [57] and PyTorch [58].

TensorFlow is developed by Google Brain team and was initially released in 2015. It has gained great popularity among researchers and developers since its debut. Some important benchmark GCNs are originally proposed and implemented using TensorFlow, such as the first version of GCN, GraphSAGE, and FastGCN. TensorFlow has a leading position in the deep learning field until PyTorch is launched.

PyTorch is a deep learning framework developed by Facebook's AI Research lab (FAIR). More recent works on graph neural networks are done using PyTorch because of its robust support for GPU related training. It defines a tensor class (torch.Tensor), which can be easily transformed to Nvidia GPU. PyTorch supports implementation of neural networks on GPU using CUDA extensions and facilitates GCN implementation with help of PyG (PyTorch Geometric Library) [26]. This opens the possibilities to take advantage of GPU computational power for GCN training.

In addition to including CUDA to its library, PyTorch provides different choices of backend support such as NVIDIA Collective Communications Library (NCCL) [17]. NCCL follows the widely used Message Passing Interface (MPI). It provides fast communications for multiple GPUs and is compatible with various multi-GPU parallelization models. For now, PyTorch only supports NCCL collective communications, such as Broadcast, AllReduce and

AllGather. Point to Point communication among GPUs can be mimicked by creating subgroups for processes.

Although using such frameworks will limit certain low-level implementation flexibility and optimizations, the advantage of its simplicity and general applicability still attract more and more users. It also accelerates research of GCN-related problems and helps to focus on the optimization of algorithms rather than low level implementations. In our research, we choose PyTorch as the base framework for its strong support of GPU-related computing and a comprehensive library for distributed and parallel deep learning training.

2.4.2 Synchronization Using All-reduce

All-reduce is a collective communication operation normally used in distributed deep learning. The all-reduce algorithm collects the target data in all workers to a single variable and returns the result to all processes in the same community. For example, there are p parallel workers, and each worker has a data D_p. Then the result of all-reduce can be represented as:

$$D_{allreduce} = D_1 \mathit{Op}\, D_2\, \mathit{Op} \ldots \mathit{Op}\, D_p ,$$

where *Op* is an operator such as SUM, MAX and MIN. An example of all-reduce call on four processors is displayed in Figure 2.8. Some important libraries such as MPI [59] and NCCL [17] have included build-in support for all-reduce call, which makes implementation for parallel GCN training more convenient.

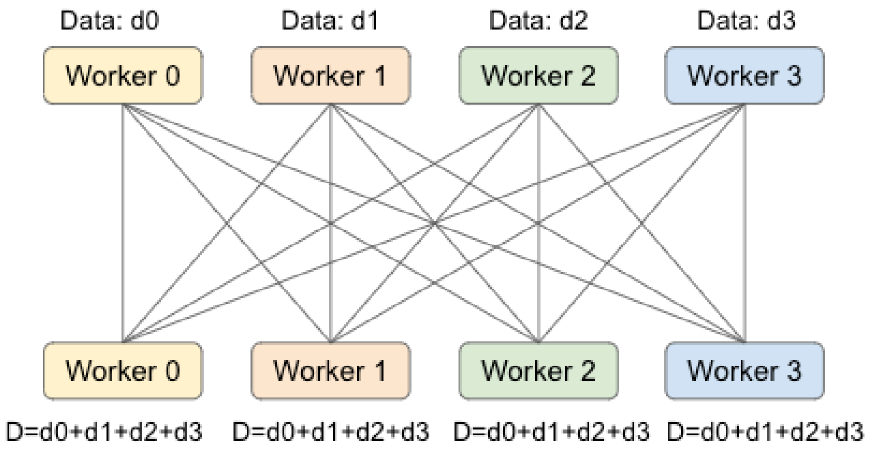


Figure 2.8 Example of all-reduce call on four processors. (Figure courtesy of [12])

The performance of synchronous GCN training is satisfying in scenarios where all workers calculate gradients at the same pace. Since there is a barrier at the end of each iteration, where all processes need to execute the same line of code to average gradients, a straggler in the system will slow down the overall training time significantly.

Efforts have been made to parallelize communication and computation during the training process to alleviate the drawback caused by the synchronous barrier. For example, PyTorch distributed data parallel (DDP) library [12] proposes a method to group gradients into several "buckets". When all gradients in the same bucket are ready, it executes gradient averaging for that portion of gradients right away, without waiting for other gradients in different buckets. Although this optimization achieves better speed up than the purely synchronous version, it still needs to wait for all gradients are averaged to end the iteration.

2.4.3 Implementation Using CPU/GPU Cluster

To cope with the problem of communication bottleneck in synchronous deep learning training using the all-reduce method, different asynchronous algorithms are proposed. Early research of parallel deep learning training on a CPU/GPU cluster normally uses CPU or a single GPU as the parameter server for asynchronous gradient averaging. However, as explained in previous sections, such centralized implementations usually suffer from speed up bottleneck at the central node. Recently, decentralized strategies have been proposed for parallel training in distributed systems, which typically use GPU for calculations and use CPU for gradient averaging.

In the work of [15], the authors propose a gossiping asynchronous decentralized algorithm for SGD training (AD-PSGD) and suggest to calculate gradients and update weights on GPU devices and execute communications on CPU. This approach helps to parallelize communication and calculation by running two separate threads on CPU and GPU respectively. Their experiments show that the proposed algorithm works well on distributed systems with 32 compute nodes with 4 GPUs on each node. With a close examination of their implementation, we notice that the asynchronous averaging only happens on the CPU level, while within each compute node, the gradients averaging among different GPUs still follows the synchronous all-reduce scheme. Another asynchronous decentralized parallel SGD approach is presented in [16]. Similarly, the asynchronous gradient averaging method is implemented for different compute nodes.

We notice in previous research, the asynchronous decentralized gradient averaging algorithms are proposed for multiple compute nodes, where asynchronous communication happens among CPUs, however, there is limited research for asynchronous gradient averaging for multiple GPUs within a single compute node. Because of the differences between CPUs and GPUs, it is not easy to apply previous methods directly to achieve asynchronous parallel training on GPUs.

Chapter 3 Graph-partition based Synchronous Data Parallel Implementation

In this chapter we first present the implementation for graph-partition based GCN using a single GPU, which sets the baseline for our research. The implementation is inspired by Cluster-GCN [8]. Then, we design and implement a synchronous GPU-parallel GCN (AllReduce-GCN). We use a random partition method to prepare mini batches and the all-reduce method to average gradients. We also illustrate an optimization method by reducing the gradient averaging frequency following the concept of periodic gradient averaging. In addition, we summarize previous theoretical discussions on the convergence and correctness of synchronous parallel training and periodic gradient averaging.

Experiments are carried out using datasets with different sizes. The results demonstrate the limitations for synchronous graph-partition based GCN. Reducing gradient averaging frequency can help to reduce communication overhead. However, because of the synchronization barrier in the training, the speed up is still limited, especially in a heterogeneous environment with workers at different paces.

3.1 Baseline Implementation for GCN Training on a Single GPU

The pseudocode for graph-partition based GCN on a single GPU is shown in Algorithm 1. In such an implementation, the input graph is first partitioned into *n* subgraphs using the random partition method. These subgraphs are used as mini batches for the training. Then a model is initialized on the GPU. During the training, one mini batch is used for gradient calculation in one iteration, and the model is updated based on the gradients. An *epoch* is reached when all the subgraphs are passed through the GCN training for one time. Usually, GCN training requires multiple epochs until the model parameters become stable. Since there is a single worker during the training, there is no need to do gradient averaging.

Algorithm 1: Baseline GCN training using a single GPU based on graph partitioning.

```
Input: Graph $G$, feature $X$, label $Y$
Output: GCN model with trained weights
    1. Partition input graph into $n$ subgraphs $G\_1, G\_2, ... G\_n$
    2. Initialize model on the GPU, define loss function, optimizer and epoch number $e$
```

```
3. for each epoch in total epochs $e$ do
4.     for each mini batch (subgraph) in subgraph_set do
5.         Get mini batch features and labels
6.         Calculate gradients
7.         Update weights
8.     end for
9. end for
```

We adapt the algorithm proposed in Cluster-GCN, but different from the original research, we use a random clustering method instead of Metis to get subgraphs. The pseudocode for the random clustering method is illustrated in Algorithm 2. To partition a graph into *n* subgraphs, we first initialize *n* empty subgraphs. Then we go through each node in the original graph and assign the node randomly to a subgraph. After all nodes are assigned, the edges for local nodes in subgraphs are retrieved from the parent graph. Then, features and labels of the nodes are copied from the original graph to subgraphs.

Algorithm 2: Graph random partition.

```
Input: Graph $G$
Output: A set of partitioned subgraphs
1. Initialize $n$ empty subgraphs
2. for node in graph G do
3.     Assign the node randomly to a subgraph
4. end for
5. for each subgraph do
6.     Build subgraph with local nodes and retrieved edges
7.     Copy node data (features and labels) from the parent graph G
8. end for
```

In our experiments, we find out that the random clustering method is much faster than Metis and produces the desired accuracy with properly chosen model and hyperparameters. One of the explanations is that the result of the training is a heuristic model and random partition avoids the problem of biased clustering. Using Metis may cause nodes of similar labels to be partitioned in the same subgraph, but random partition can ensure the same probability of nodes appearing in any subgraph. Another reason is that the authors of the original paper only conduct experiments using the original version of GCN, which has a relatively lower accuracy compared to other advanced models (e.g. GraphSage). We conduct the experiments based on the open sourced code examples in DGL GitHub [33] by using GraphSage aggregator. The accuracy improved by using a superior model can minimize the variance brought by different clustering methods. Considering the time consumption for the graph partition phase, we decide to use the random partition method in our research.

By using graph-partition based GCN, the mini batch preparation phase and the "real-training" phase (gradient calculation and weight update) are separated. In our research, we compare the later one ("real-training" phase) for different algorithms. Since the graph partition phase is the same for different implementations discussed in this work, the graph partition time is not included when comparing speed up.

3.2 Synchronous GPU Parallel Implementations (AllReduce-GCN)

In this section, we present a decentralized synchronous GPU-parallel algorithm (AllReduce-GCN) for graph-partition based GCN. We elaborate the implementation details and an optimization method of reducing synchronization frequency. Previous theoretical discussions for the related algorithms are presented in this section. Experimental results and related datasets details are also provided.

3.2.1 Synchronous GPU Parallel using All-reduce

In a decentralized graph-partition based GCN training, the input graph is first partitioned into *n* subgraphs and these subgraphs are used as sample mini batches for the training. Each GPU in a compute node is a worker assigned with a mini batch. Each worker initializes a local model and gets n/p portions of subgraphs (mini batches). During the training, at each iteration every worker processes a mini batch locally and then averages local gradients with all other workers for updating the model parameters. Different from the baseline GCN training on a single GPU, where gradient averaging is done after processing each mini batch once, in parallel GCN training the gradient averaging is done after processing every p mini batches; as a result, there is a difference in accuracy. The gradient averaging step is usually done by using a reduction operator in all-reduce which synchronizes among all the workers. Same as the baseline training on a single GPU, it requires multiple epochs until the model parameters become stable. The pseudocode of a decentralized synchronous GPU parallel implementation for graph-partition based GCN training is illustrated in Algorithm 3.

Algorithm 3: Synchronous GPU-parallel GCN training based on graph partitioning

```
Input: Graph G, feature X, label Y
Output: GCN model with trained weights
1.  Partition input graph into n subgraphs G_1, G_2, … G_n
2.  Start p workers, and each GPU is assigned with 1 worker
3.  Initiate model on each worker, define loss function, optimizer and epoch number e
4.  Divide subgraphs evenly among workers, with n/p subgraphs in each worker's
    subgraph_set
5.  /* Each worker does the following */
6.  for each epoch in total epochs e do
7.      for each mini batch (subgraph) in subgraph_set do
8.          Get mini batch features and labels
9.          Calculate local gradients
10.         Average gradients with other workers using all-reduce
11.         Update weights
12.     end for
13. end for
```

As shown in Algorithm 3, lines 1 to 4 demonstrate the preparation work before the training starts. The epoch number *e* is determined empirically and is set before the training starts. The instructions starting from line 5 to the end are executed in each worker. In each iteration, one mini batch is taken from the *subgraph_set* and passed to the local model stored in GPU for training. Lines 8 and 9 get subgraph (mini batch) features and labels from the CPU, and the data is used to calculate local gradients using forward and backward propagation. In line 10, local gradients are averaged with all the workers using the all-reduce method. Then in line 11, the model parameters are updated using the averaged gradients. The training is finished when the required epochs are passed.

Although it is possible to map multiple threads on a single GPU if the model is small enough, the communication speed of all-reduce call is affected greatly because of the competition for GPU bandwidth. In addition, when a NCCL collective communication (e.g. all-reduce) is running, it blocks other jobs and waits for other peers to join, which can be a bottleneck for overall training speed. Moreover, if the GPU is busy and doesn't have enough free computing ability for all jobs, it will store the jobs in a stack and take one to execute whenever it is free. So even with multiple threads mapped to one GPU, the mini batches may get trained in a sequential manner in such a scenario. Considering all these effects, practically we map only one thread with each GPU.

Figure 3.1 shows the overall scheme of the implementation for synchronous graph-partition based GCN using the all-reduce method (AllReduce-GCN). The subgraphs (mini batches) are prepared in advance using the same random partition method as for the baseline counterpart using one GPU, which is illustrated in the previous section. During the training, the features and labels of nodes in the training sample are sent to GPU, so the memory consumption can be estimated using the size of the samples. For each iteration, gradients are calculated in parallel, and the optimization is done by PyTorch backend implementation.

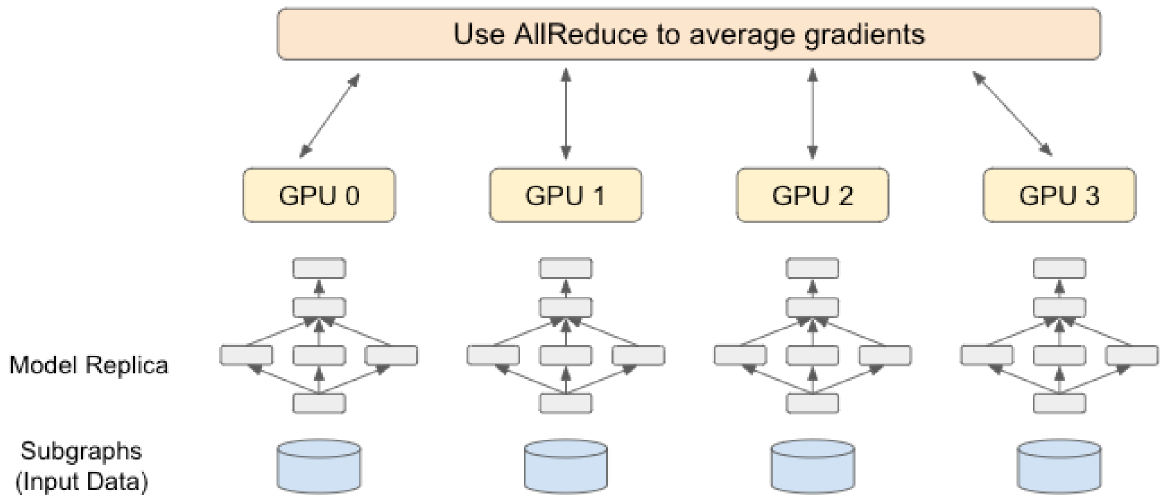


Figure 3.1 Scheme of synchronous graph-partition based GCN using all-reduce.

In the parallel implementation, a training sample in one worker is also called a mini batch, and the collection of mini batches on all workers in one iteration is called a global batch. Since the gradients are averaged at each step, actually the model is updated based on a larger batch compared with the baseline implementation. For example, the batch size for the baseline version is x, and in the parallel implementation there are p workers, the batch size for each iteration is changed to px. In another word, the baseline GCN implementation trains each mini batch one by one and uses the previous gradients to update parameters of the model. So, each mini batch is trained based on information gathered from previous samples. While in parallel GCN, multiple mini batches are trained simultaneously. These differences have influence on the final results for the model and lead to variance in accuracy.

To have a fair comparison for the speed up, we use the same batch size for the baseline and parallel implementations, so the total amount of training work is identical for different algorithms. But since gradient transmission and averaging need to take extra time, it adds to the overall workload and time consumption for parallel GCN. We choose the refined hyperparameters for the baseline implementation that give the best results and use the same hyperparameters for parallel implementations. Experiments show that parallel GCN training has a better speed up than the baseline one. However, according to the reasons stated above, the accuracy of parallel GCN training is affected. So, there is a trade-off for accuracy and training speed in practice.

PyTorch has an optimization to parallelize communication and computation in each iteration with its DistributedDataParallel module. We have compared the optimized version with the normal implementations, the training speeds have a small difference for the tested datasets. Because the major bottleneck is the averaging gradient in each step, and even with the optimization, it still needs to wait for all parallel workers to finish the calculation and average gradients. In Figure 3.2, we illustrate how the barrier of synchronization causes faster workers to stay idle and adds up to communication overhead.

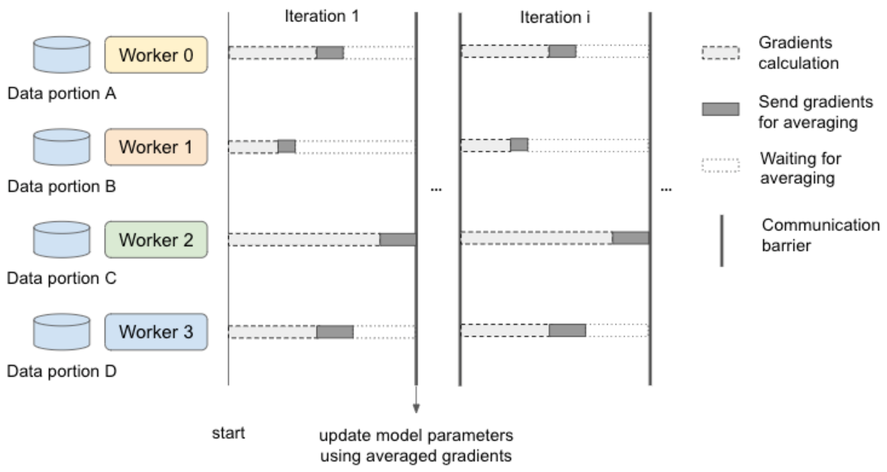


Figure 3.2 Synchronous data parallel AllReduce-GCN. (Figure courtesy of [60])

3.2.2 Optimization with Periodic Gradient Averaging

As discussed in Chapter 2, there are various methods to mitigate the consequences caused by the communication bottleneck. The periodic gradient averaging is discussed in previous research to reduce communication overhead. In this section, we propose an optimized implementation for AllReduce-GCN by reducing synchronization frequency.

Inspired by the concept of local SGD [53] and model averaging [56], we implement a periodic gradient averaging optimization for the AllReduce-GCN. We define an *averaging_interval* to control gradient averaging frequency and initiate a variable *count* to track the number of mini batches trained in the workers. Instead of executing the synchronization in each step, we allow the parallel workers to train multiple mini batches locally and perform synchronization in a certain interval. The modified pseudocode for AllReduce-GCN is presented in Algorithm 4.

Algorithm 4: Optimization for AllReduce-GCN using periodic gradient averaging

```
Input: Graph G, feature X, label Y
Output: GCN model with trained weights
1.  Partition input graph into n subgraphs G_1, G_2, … G_n
2.  Start p workers, and each GPU is assigned with 1 worker
3.  Initiate model on each worker, define loss function, optimizer and epoch number e
4.  Define averaging_interval, and initiate count with 0
5.  Divide subgraphs evenly among workers, with n/p subgraphs in each worker's
    subgraph_set
6.  /* Each worker does the following */
7.  for each epoch in total epochs e do
8.      for each mini batch (subgraph) in subgraph_set do
9.          Increment count by 1
10.         Get mini batch features and labels
11.         Calculate local gradients
12.         if count is multiple of averaging_interval then
13.             Average gradients with other workers using all-reduce
14.         Update weights
15.     end for
```

```
16. end for
```

Since gradients are averaged periodically, the time for gradient transmission and averaging calculation is reduced. However, the speed up achievement is still limited by the synchronization barrier as shown in Figure 3.3. All parallel workers need to pass the same number of iterations until reaching the gradient averaging point. Since it is hard to guarantee that all GPUs can execute the jobs at the same pace, if there is a slower worker, other faster workers need to wait for the slower worker to catch up. This implementation doesn't take full advantage of the computing capability for fast workers, because the workload is divided evenly for all GPUs regardless of their available computing capacity.

Considering this situation, we find it is not a superior strategy to divide the amount of work in a naive data parallel fashion. A solution for this is to assign mini batches dynamically in the training process, whenever a worker finishes a current job, it can request for the next mini batch. This work-pool scheme can reduce the idling time for faster workers. However, this strategy is not compatible with synchronous GPU-parallel implementation using the all-reduce method. Since all workers need to average gradients together, it requires the parallel worker to train the same number of samples at the end. To balance the workloads among different workers, we propose to have a work pool mechanism in an asynchronous GCN training algorithm, which is explained in Chapter 4.

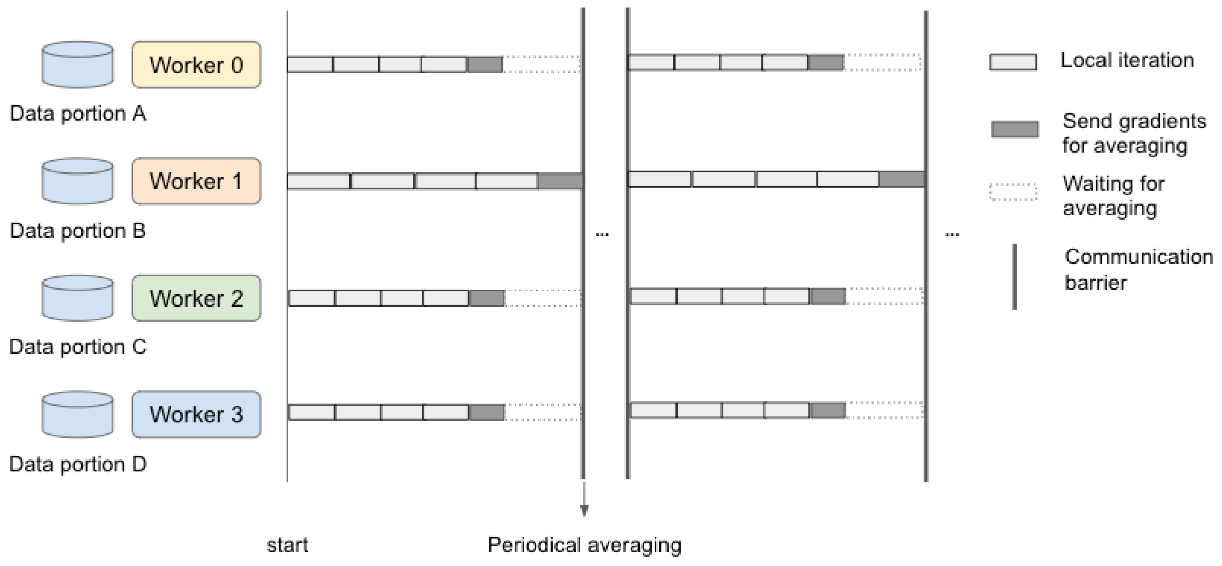


Figure 3.3 Barrier of AllReduce-GCN with periodic synchronization.

3.2.3 Theoretical Discussion

In this section, we conclude previous theoretical discussions for the related algorithms of AllReduce-GCN. The topics include convergence and correctness of the baseline Cluster-GCN on a single GPU, synchronous data parallel training and periodic gradient averaging.

A theoretical discussion on baseline ClusterGCN is already presented in [8]. The ClusterGCN algorithm is inspired by mini-batch SGD, which computes gradients based on a mini batch in each iteration. The baseline ClusterGCN algorithm follows similar methodologies and convergence steps for SGD, however with a different approach to sample mini batches. A detailed comparison can be found in the original paper and is beyond the scope of this research.

For synchronous data parallel training, theoretical discussions are carried out in previous study [61]. A formal mathematical proof for the algorithm can be found in the paper. The general idea is that the converge can be achieved by averaging stochastic gradients computed at the same predictor. With synchronous parallel implementations, each compute node keeps a copy of the same model and the weights are updated with same averaged gradients, so it is identical to have a global model that takes training results of all mini batches. [62] summarizes a proof strategy for data parallel training in the paper. They show that since all compute nodes perform SGD training based on the same data distribution, with a fixed and small enough learning rate, models converge to the same limit. The averaging of gradients reduces variance is also proved.

Previous works [53, 56] have analysed the effect of periodical averaging of gradients at certain intervals (i.e. averaging interval discussed in section 3.3). Those studies show that periodical synchronization can lead to the same convergence rate as of performing gradient averaging at each iteration. Detailed proofs and results can be found in those works. It has been shown that if the synchronization interval is selected properly, then the divergence among different workers can be controlled and the models finally converge to the same local minimum.

3.3 Experimental Evaluations

In this section, we provide information for the experiment setup and datasets used in our research. We also illustrate experimental results for the baseline implementation using a single GPU and synchronous GPU-parallel implementation for graph-partition based GCN

(AllReduce-GCN). In addition, we test the AllReduce-GCN with different gradient averaging intervals and analyze the results.

3.3.1 Setup and Datasets

The initial implementation of Cluster-GCN is provided by the author of the original paper [8]. They propose to use the Metis graph clustering method and the original version of graph convolutional network with diagonal enhancement. DGL GitHub [33] provides another version of Cluster-GCN implementation using GraphSAGE aggregator and ignoring diagonal enhancement. Although different in details, both implementations achieve the reported F1 score 96.6 for Reddit datasets. Our baseline implementation is based on the open-sourced sample codes discussed above.

The experiments are carried out for the task of node classification using two benchmark datasets (Reddit and Amazon datasets). Table 3.1 provides the detailed information of the two datasets. In the table, “s” stands for single-label classification, and “m” stands for multi-label classification. The two datasets are used for multi-class problems where the class number is more than two. The difference for the two datasets is that in single-label classification, each node is classified into a unique class, while in multi-label classification, a node can be categorized into any number of classes. Both datasets are used widely in GCN research as benchmarks to compare performances of different algorithms. Thus, we use them in our research to test the implemented algorithms.

Reddit dataset is originally created in the research of GraphSAGE [23] by gathering post data on Reddit online discussion forum. It predicts the community that a post belongs to. Nodes in the graph represent posts, and edges between two nodes stand for the connection that the same user comments on both posts. Features are collected concatenating the embedding of post titles, comments, scores and the numbers of comments. Labels stand for the communities of posts. As shown in table 3.1, Reddit dataset has 41 classes in total and each node belongs to one class.

Amazon dataset is created in the study of GraphSAINT [9]. The task is to predict the product categories for the items. Nodes in the graph are products listed on Amazon website. Edges between the nodes means that the same person buys both products. Features are collected and organized to represent the reviews of the products. Labels stand for the categories of the products. In this dataset, a product can belong to more than one category. The authors of GraphSAINT have cleaned the database by removing rare categories that have a few products.

The total number of classes for this dataset is 107. We use the same datasets as they present in the original paper.

Table 3.1 Datasets for experimental evaluations

Dataset	Nodes	Edges	Feature	Classes	Train/Val/Test
Reddit	232,965	114,848,857	602	41 (s)	0.66/0.10/0.24
Amazon	1,598,960	132,169,734	200	107 (m)	0.80/0.10/0.10

In the following content, we illustrate the setup for our experiments. We evaluate the baseline GCN and AllReduce-GCN on a GPU cluster. The GPU cluster has a submit node, a master node and two compute nodes. Each compute node has 72C CPU, 450 GB of RAM, 7 TB of storage space, and 8 GPUs (Tesla V100). In our experiments, we use one compute node of this cluster. Since the cluster is shared by students and researchers, it is unavoidable to have multiple jobs submitted to the GPU cluster at the same time. Because of the different workloads for GPUs at different time periods, the time consumptions for a same training job vary a lot in different tests. This is a real-world scenario when training GCNs on a GPU cluster. We take records of all tests and calculate the averaged values for evaluation.

The algorithms are implemented using PyTorch DistributedDataParallel [12] and Deep Graph Library (DGL) [27] as our deep learning framework. As explained in Chapter 2, PyTorch has included CUDA and NCCL to facilitate deep learning training on GPUs, and PyTorch Geometric Library (PyG) [26] makes it more convenient to implement graph neural networks. DGL is a Python package for deep learning on graphs and has gained popularity in recent years. Anaconda is used to create the testing environment. Table 3.2 shows the versions of related frameworks and libraries.

Table 3.2 Frameworks and libraries

Name	Version
Python	Python 3.8.5
CUDA	CUDA 10.1
PyTorch	pytorch==1.5.1, torchvision==0.6.1, cudatoolkit=10.1
PyG	torch-geometric and dependencies for torch-1.5.0
DGL	dgl-cuda10.1<0.5

We use torch.distributed.all_reduce to achieve the synchronous gradient averaging in GCN training. Since our implementation is based on graph partition to get mini batches, it is unfair to compare with the methods implemented with traditional sampling methods. So, we take the implementation of graph-partition based GCN using a single GPU as the baseline for speed up calculations. Times are measured for the training phase, including gradient averaging time. Because of the different label categorization strategies of the two datasets, we use different loss functions for them. For Reddit, we use torch.nn.CrossEntropyLoss(), and for Amazon dataset we use torch.nn.BCEWithLogitsLoss(). Since the graph partition phase can be done before the real training phase, and the partition time is the same regardless of the training strategies, we store the partitioned subgraphs in a place and exclude the partition time when comparing speed ups. Table 3.3 shows the hyperparameters used for the baseline and parallel experiments on the two datasets. F1 scores are used to evaluate the accuracies of the final models.

Table 3.3 Hyperparameters used for experiments

Dataset name	Reddit	Amazon
Subgraph number	40	640
Learning rate	1e-2	1e-2
Epoch number	30	30
Hidden GCN Layer	2	2
Hidden GCN units	128	512
Dropout	0.2	0.1

3.3.2 Experimental Results

We carry out experiments for the two datasets in different time periods. The results show that with the same hyperparameters and subgraphs (mini batches), the baseline GCN training can result in the same solution (same F1 score). However, the time consumptions are very different in various scenarios. With different subgraphs (mini batches), baseline training can lead to a similar solution with minor difference in accuracy.

Cluster-GCN with random partitioning which uses a single GPU is used as the baseline for performance and accuracy evaluations. We observe from experiments that the training speed depends largely on the status of the GPU cluster. The synchronous AllReduce-GCN with gradient averaging at each iteration gives a disappointing outcome in some scenarios. We can see from the experiments that it takes even longer time for parallel training on multiple GPUs than training on a single GPU in some tests. This is because the fast workers need to wait for the slowest worker in each iteration for averaging gradients.

Although the GPU models are of same make (Tesla V100), they work at different pace if the computation competency is not large enough to execute all jobs at the same time. In most scenarios, the GPUs receive multiple jobs from different users. Based on the free calculation ability, a GPU may add the job to a stack if it cannot execute it immediately. Once it has free resources, it will take a job from the stack and run it. So, this leads to different execution times for the same job at different runs.

In addition, the collective GPU communication has an effect on the training speed for AllReduce-GCN. In synchronous AllReduce-GCN, there is only one all-reduce call at one time among all the workers. However, we cannot make sure that other jobs running on the GPU cluster don't require GPU communications at the same time. If there are multiple GPU communication calls, the jobs will compete for the bandwidth among GPUs. This will cause a delayed training time for synchronous GCN training using the all-reduce method. We observe this time variation in our tests for AllReduce-GCN using 4 and 8 GPUs.

Compared with baseline GCN training, parallel training has an accuracy loss, the main reason is that parallel training of graph-partition based GCN uses a different batch size for weight updates. During the tests, we use the same hyperparameters for both baseline and parallel training. The hyperparameters are tuned for baseline training, so it can produce an optimal solution when using a single GPU.

Since the hyperparameters are the same, the mini batch size remains same for the baseline and parallel training. In parallel training, one mini batch is trained at each worker simultaneously, then the gradients are averaged, and the weight is updated using the averaged gradients. So, the update in each iteration is actually based on information of mini batches trained at all workers. The total mini batches used for one iteration is called a global batch. The different global batch sizes influence the training behaviour and lead to different final models. This explains why we observe different accuracies for parallel training on multiple GPUs.

Figure 3.4 shows speed up for Reddit and Amazon dataset using different numbers of GPUs. The values used in the figure are illustrated in Tables 3.4 and 3.5. Since the training times vary in different runs, we use the averaged values to produce the figures in this section.

Table 3.4 Speed up of AllReduce-GCN for Reddit dataset.

GPU number	1	4	8
Speed up	-	1.87	2.93
F1 score	0.9655	0.9637	0.9609

Table 3.5 Speed up of AllReduce-GCN for Amazon dataset

GPU number	1	4	8
Speed up	-	1.74	2.81
F1 score	0.7828	0.7745	0.7721

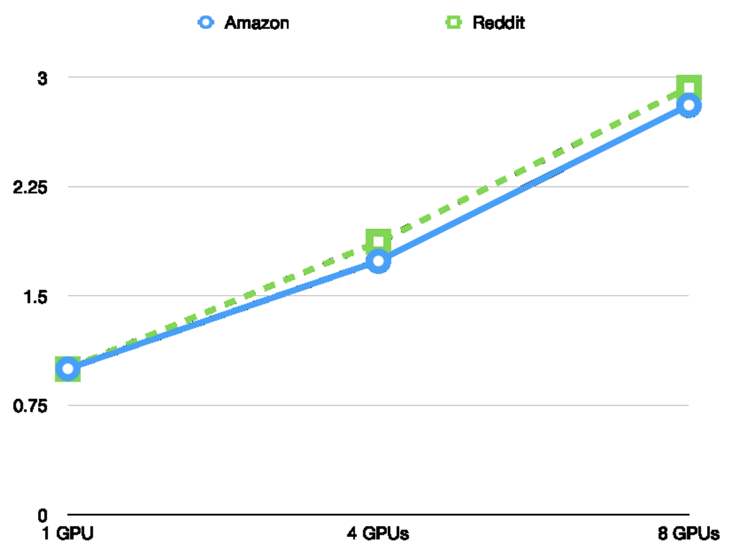


Figure 3.4 Speed up of AllReduce-GCN for Reddit and Amazon dataset.

The sublinear speed up shown above is mainly because of the synchronization barrier and communication overhead. As explained in the synchronous AllReduce-GCN algorithm, there is a barrier at the end of each iteration where all workers need to communicate and average gradients. If there is a slow worker, the fast workers need to wait for the gradient averaging and stay idle. So, the speed up of the system is limited by the slowest worker. In some extreme situations, it may be even slower to use multiple GPUs than using a single fast

GPU. In addition, the transmission of gradients causes extra communication overhead. When the size of the gradients is small, it may have little influence on the speed up, but when the size is large, it may result in longer training time.

We also carry out tests using different gradient averaging intervals for the synchronous AllReduce-GCN. According to empirical experience, we test performing gradient averaging in every 20 and 100 iterations. Tables 3.6 and 3.7 show the results of synchronous AllReduce-GCN training for Reddit dataset with different synchronization intervals (averaging_interval) on 4 and 8 GPUs. Table 3.8 and 3.9 show the experimental results for Amazon dataset.

Table 3.6 AllReduce-GCN for Reddit dataset (4 GPUs)

Averaging_interval	1	20	100
Time / sec	16.01	14.08	13.66
F1 score	0.9637	0.9501	0.9508

Table 3.7 AllReduce-GCN for Reddit dataset (8 GPUs)

Averaging_interval	1	20	100
Time / sec	10.21	10.11	10.07
F1 score	0.9609	0.9449	0.9435

Table 3.8 AllReduce-GCN for Amazon dataset (4 GPUs)

Averaging_interval	1	20	100
Time / sec	103.65	87.13	79.31
F1 score	0.7745	0.7473	0.7472

Table 3.9 AllReduce-GCN for Amazon dataset (8 GPUs)

Averaging_interval	1	20	100
Time / sec	64.18	59.13	53.20
F1 score	0.7721	0.7189	0.7187

Figure 3.5 and 3.6 illustrate the comparisons of synchronous AllReduce-GCN training for Reddit and Amazon datasets using different synchronization intervals. The values used for producing the figures are elaborated in Table 3.10 and 3.11.

Table 3.10 Speed up of AllReduce-GCN for Reddit Dataset

GPU number		1	4	8
Speed up	averaging_interval = 1	-	1.87	2.93
	averaging_interval = 20	-	2.13	2.96
	averaging_interval = 100	-	2.19	2.97

Table 3.11 Speed up of AllReduce-GCN for Amazon Dataset

GPU number		1	4	8
Speed up	averaging_interval = 1	-	1.74	2.81
	averaging_interval = 20	-	2.07	3.05
	averaging_interval = 100	-	2.27	3.39

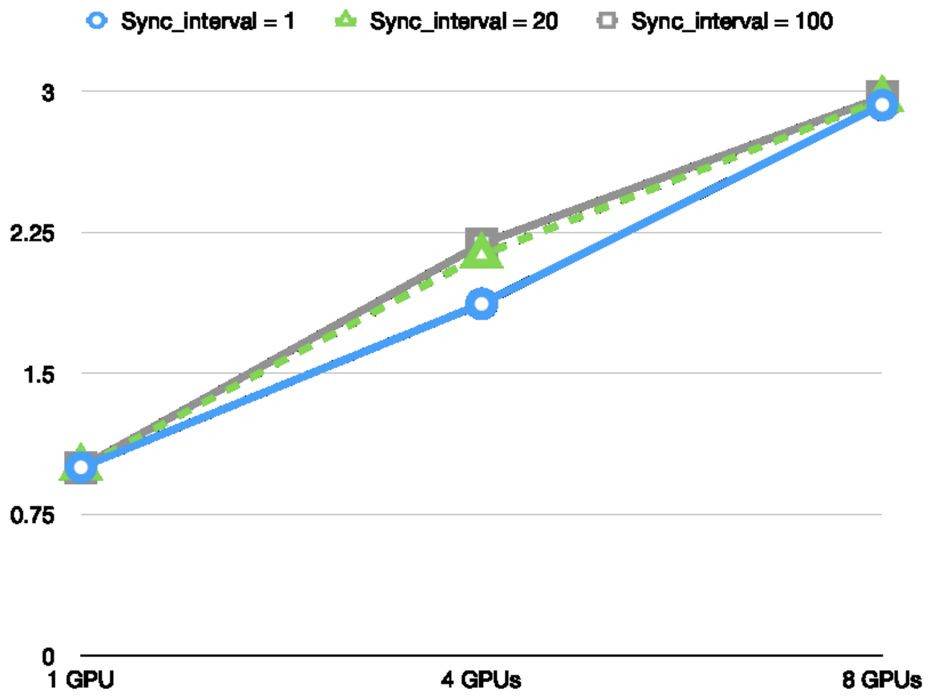


Figure 3.5 Speed up of AllReduce-GCN for Reddit Dataset.

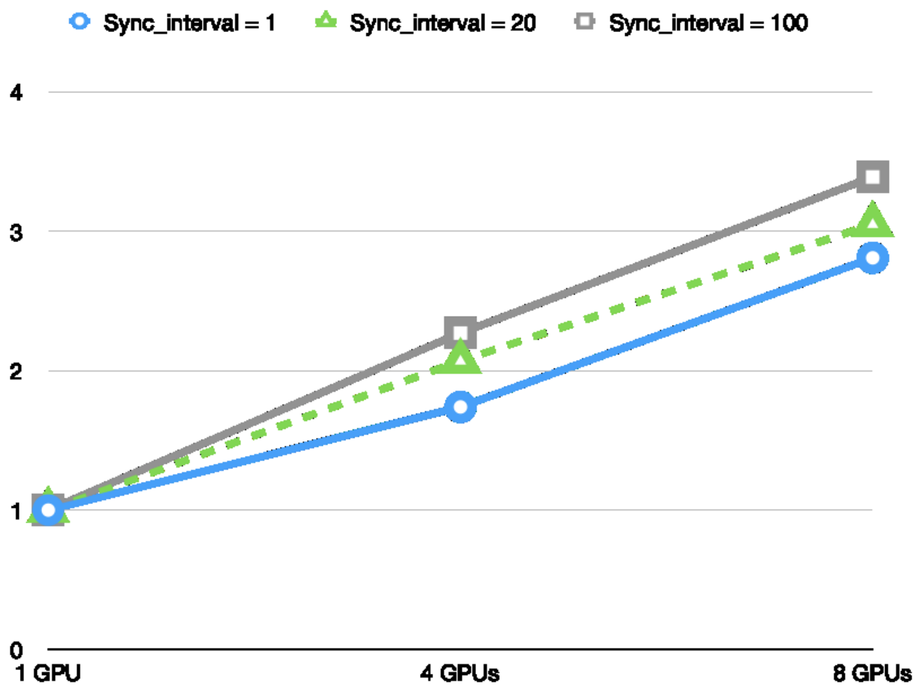


Figure 3.6 Speed up of AllReduce-GCN for Amazon Dataset.

We can see from the above figures that reducing synchronization frequency can help to achieve a bit more speed up, especially when the size of gradients is very large. This is because by reducing the averaging frequency, time is saved for gradient transmission among GPUs. However, the speed up achievement is still limited by the communication barrier. In the optimization implementation, the gradient averaging occurs in a certain interval of local iterations. This means that the numbers of mini batches trained locally in the parallel workers are identical. So, fast workers still need to wait for slower ones to catch up and average gradients.

It is already known from previous research that one-shot averaging can lead to low accuracy of the final model. In one-shot averaging, gradients are only synchronized for one time at the end of the training. In our experiments, we also observe that the synchronization frequency has an influence on the accuracy of the final solution. So, there is a trade-off for accuracy and speed up.

3.4 Summary and Limitations of AllReduce-GCN

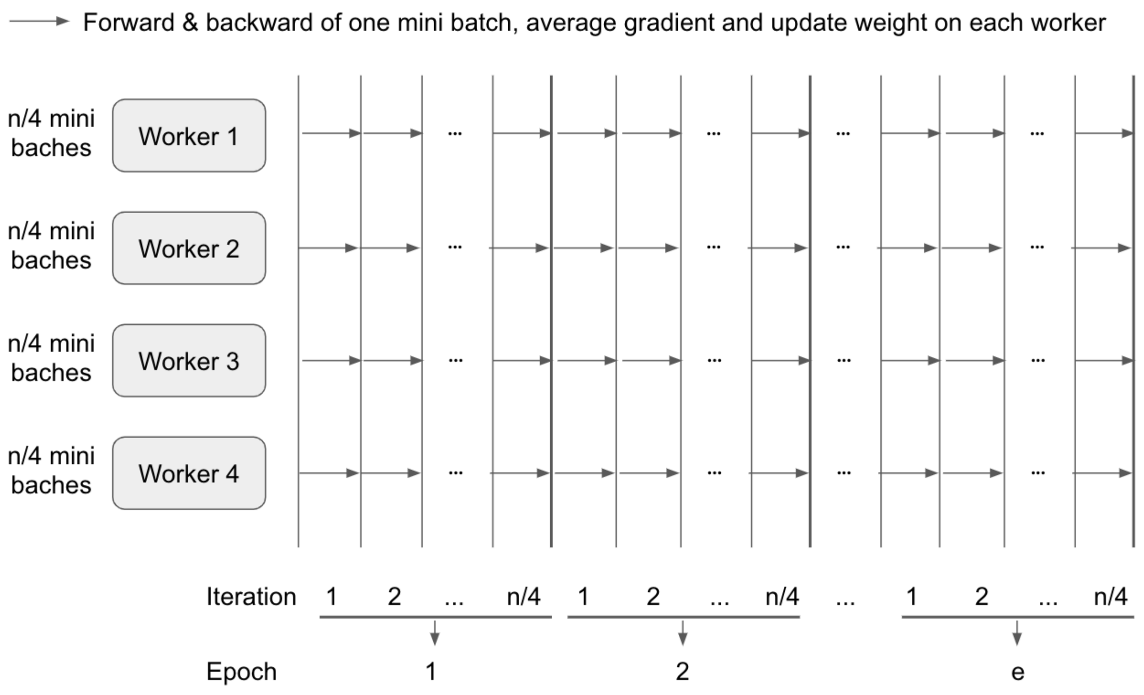
In recent years, communication among GPUs has become more convenient with the development of GPU backend platforms like NCCL [17]. To get rid of the bottleneck of a centralized parameter server and passing messages between GPUs and CPUs, a decentralized parallel stochastic gradient descent algorithm is widely adopted in current implementations of parallel GCNs training with multiple GPUs.

In our research, we implement a synchronous decentralized data parallel GCN based on graph partition method (AllReduce-GCN). We focus on the speed up for the training phase. We also implement an optimization for AllReduce-GCN by using periodic gradient averaging.

In such an implementation, each worker synchronizes with other workers using all-reduce at the end of processing a mini batch. The synchronization step works well with the assumption that each GPU computes the gradients at the same pace. However, in a real-world scenario, this assumption is not valid due to two reasons: the compute node can be busy with multiple jobs; secondly, partition of the input graph cannot guarantee exactly the same size for partitioned subgraphs, which can lead to load imbalances. Our experiments on a GPU cluster with Tesla V100 GPUs illustrate the load-imbalance and resultant idling in a realistic situation. Although PyTorch supports optimizations of parallel communication among GPUs with the DistributedDataParallel module, it does not solve the problem of synchronization delays due to idling [12].

Another bottleneck with algorithm 3 and 4 is that the mini batches are divided statically among the workers, which adds up the workload differences across multiple iterations. With the optimization method of periodic gradient averaging, it helps a bit to improve the speed up by reducing the time of gradient transmission, but it cannot solve the major communication bottleneck caused by synchronous training. An example of such work division on 4 workers is shown in Figure 3.7.

To address the previous issues of bottleneck in the synchronous implementations, we propose an asynchronous training algorithm (GossipGCN) based on gossiping designed for GPUs. GossipGCN is motivated by limitations of traditional synchronous GCN training algorithms on GPUs. The algorithm incorporates dynamic workload assignment based on a work-pool strategy. In the following chapter, we describe the proposed asynchronous GPU parallel implementations in detail.


Forward & backward of one mini batch, average gradient and update weight on each worker
n/4 mini baches
Worker 1
n/4 mini baches
Worker 2
n/4 mini baches
Worker 3
n/4 mini baches
Worker 4
Iteration
1 2 ... n/4 1 2 ... n/4 ... 1 2 ... n/4
Epoch
1
2
e
Figure 3.7 Fixed workload divide mechanism for AllReduce-GCN.

Chapter 4 Graph-partition based Asynchronous Data Parallel Implementation

In this chapter, we present an asynchronous parallel algorithm for GCN training called GossipGCN. The algorithm employs graph partitioning as a mechanism for preparing the training mini batches and (asynchronous) gossiping for periodic averaging of gradients among randomly paired workers (GPUs). The methodology proposed in Cluster-GCN [8] is adapted. The algorithm is proven to be deadlock-free. A work-pool based mechanism is used for dynamic assignment of workloads to workers (GPUs), which enhances efficiency by reducing idling. The algorithm is implemented on a GPU cluster using PyTorch [58] and DGL [27]. The details are in the following.

4.1 Asynchronous Training based on Gossip

GossipGCN is motivated by limitations of traditional synchronous GCN training algorithms on GPUs. In chapter 3, we have discussed why synchronous GPU-parallel implementation for GCN has a speed up limitation because of the averaging gradient among all workers. To improve speed up of parallel training on a single node, we propose to use decentralized gossip algorithm to average gradients among GPUs and implement a work-pool based mechanism to maximize the usage of computation ability of fast workers.

Gossip is used to solve consensus problems and is widely used in many application areas, especially in distributed environments. Previous research [14-16] has shown that gossip is an effective method for asynchronous gradient averaging in SGD-based CNN training in a distributed environment. In such a distributed environment, the gradient calculation and weight updates are usually carried out on GPUs while the asynchronous communication for averaging gradients runs on CPUs on a multi-CPU platform. Until now, gossip methods are mainly used for synchronization among compute nodes using CPUs, applying gossip algorithm to average gradients inside a compute node with multiple GPUs is not well studied yet.

One major challenge of gossip is to effectively and efficiently prevent deadlock. Deadlock can occur when the dependencies of workers result in a cycle. For example, worker A randomly picks a gossip partner B, while B is waiting to partner with C, and C sends a request to A. In this case, there is a cyclic dependency which is a necessary condition for deadlock. Previous works propose different solutions to prevent deadlocks in gossiping. Figure

4.1 shows the scheme of random gossiping and an example of deadlock. In [50], a request-based protocol is used to avoid cyclic dependencies. GossipGraD [14] introduces a virtual organization of compute nodes so that deadlock is not possible. In [15], the workers are divided into two fixed sets, the active and passive sets. A worker in the active set can pick a gossip partner only from the passive set, thus essentially preventing a cycle and hence deadlock.

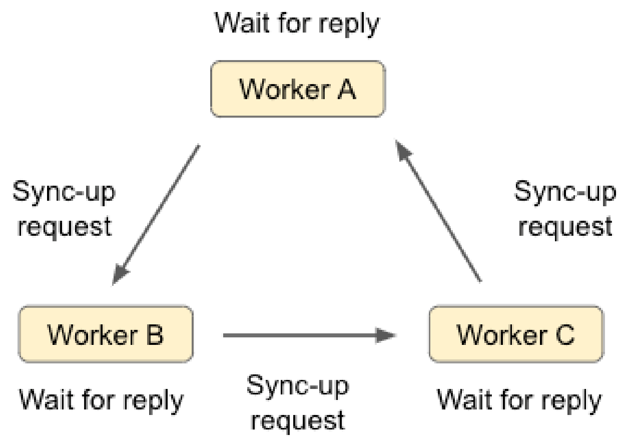


Figure 4.1 Deadlock in gossip averaging.

Even though the previous solutions can guarantee deadlock prevention, they are not suitable for asynchronous GCN training on GPUs due to the following reasons. The request-based solution needs to send additional requests in order to determine gossip partners, which adds up to communication overhead in a communication-constrained GPU backend. GossipGraD imposes a fixed gossip scheduling, which is not efficient in a dynamic system with stragglers, as in GCN training. The solution based on fixed active-passive sets puts restrictions on who can select who as a gossip partner and thus it is only a pseudo implementation of the original randomized gossip algorithm.

Another challenge for implementing decentralized gossip method among GPUs is to achieve one to one communication efficiently. According to NCCL documentation, since NCCL 2.7 point-to-point communication can be achieved using ncclSend and ncclRecv. However, in PyTorch, receive and send between two specific GPUs are not supported yet. We propose a workaround method by using subgrouping method to achieve point-to-point communication among GPUs.

4.2 Asynchronous GPU Parallel Implementation Version 1

In a preliminary attempt, we try to implement asynchronous GPU parallel training by adapting previous methods to solve the gossip deadlock problem. For example, we try to send pre-requests to workers before the real gossip happens. However, the acknowledging and replying process takes a long time. This adds to the communication overhead, so it is not a proper way to implement gossip for GPUs. We also try to use scheduled paring with a fixed gossip pattern. But since the gossip partners are fixed, there is information loss during the training process. This method can't guarantee a satisfying accuracy.

One working method that we implement for asynchronous GPU parallel training is inspired by the approach used in [15]. The paper proposes to use the property of bipartite graph to avoid deadlocks. It suggests dividing the workers into two groups, specifically active set and passive set. Workers in the active set can pick a worker only from the passive set for synchronization. As explained in Chapter 2, the original paper implements gossip communication only for CPUs with traditional sampling methods. In our approach, different from the original implementation, we achieve gossip among GPUs and use graph-partition based GCN.

The following are the main steps of the proposed asynchronous algorithm version 1. First, we partition the input graph into *n* subgraphs, which are used as mini batches. Then the epoch number and averaging interval are decided empirically, and a shared queue is initialized with mini batch ids to function as a work-pool. Suppose there are an even number of parallel workers, we set half of them as active workers and the other half as passive workers. Available passive worker ids are stored in a shared queue. And each passive worker is assigned a shared variable to store its gossip partner. When the training starts, each worker takes a mini batch from the work-pool and calculates local gradients. Then for an active worker, if the averaging interval is met, it picks a gossip partner from the available worker list and tells the corresponding passive worker through the shared variable assigned to it. For a passive worker, it reads from its corresponding variable to get the gossip partner id and average gradients with the partner. When the averaging is finished, each worker updates its local model and requests one more mini batch from the work-pool until all jobs are done.

Algorithm 5 shows the pseudocode for asynchronous GPU parallel implementation version 1 with active and passive sets of workers.

Algorithm 5: Gossip gradient averaging using active & passive worker sets

```
Input: Graph G, feature X, label Y
Output: GCN model with trained weights
      /* Master does the following from lines 1-8 */`
  1.  Partition input graph into n subgraphs G_0, G_1, ... G_n-1
  2.  Set epoch number as e, a pre-set value determined empirically
  3.  Initialize work_pool = [0,1, ... n-1, 0, 1, ... n-1, ... 0, 1, ... n-1]. /* The shared
      work_pool queue stores the mini batch indexes. Each index is stored e times */
  4.  Separate p workers into active and passive sets, each with size of p/2
  5.  Initialize shared variable available_workers with passive worker ids
      available_workers = [0, 1, ... p/2-1]
  6.  Initialize shared variables p_0, p_1, ... p_p/2-1 with -1 for passive workers
      /* These variables are used to store each passive worker's gossip partner id. -1
      represents not paired */
  7.  Start p workers, and each GPU is mapped with 1 worker
  8.  Initialize model on each GPU
  9.  Initialize averaging_interval
  10. if proc_id belongs to active set then
  11.     /* Each active worker does the following from line 12-29 */
  12.     Initialize count = 0
  13.     Initialize partner = -1
  14.     while work_pool is not empty do
  15.         Remove a mini batch index i from work_pool
  16.         Increment count by 1
  17.         Get G_i features and labels from CPU
  18.         Calculate local gradients using G_i
  19.         if count is a multiple of averaging_interval then
  20.             Lock available_workers
  21.             if available_workers not empty then
  22.                 partner = randomly choose an id x from available_workers
  23.                 Set corresponding partner's gossip id p_x to proc_id
  24.                 remove partner from available_workers
```

```
25.                 Unlock available_workers
26.                 if partner > -1 then
27.                         Average gradients with partner
28.                         Reset partner = -1
29.             Update weights for local model
30. else
31.     /* Each passive worker does the following from line 32-54 */
32.     while work_pool is not empty do
33.             Set local variable partner to -1
34.             Remove a mini batch index i from work_pool
35.             Get G_i features and labels from CPU
36.             Calculate local gradients using G_i
37.             Lock the corresponding variable p_proc_id
38.             Set partner = p_proc_id
39.             Set p_proc_id value to -1 and unlock
40.             if partner > -1 then
41.                     Average gradients with partner
42.                     Lock available_workers
43.                     Add proc_id back to available_workers
44.                     Unlock available_workers
45.             Update weights for local model
46.     /* Clean-up step */
47.     Lock available_workers
48.     Remove proc_id back from available_workers
49.     Unlock available_workers.
50.     Lock the corresponding variable p_proc_id
51.     Set partner = p_proc_id
52.     Set p_proc_id value to -1 and unlock
53.     if partner > -1 then
54.             Average gradients with partner
```

This implementation makes sure that a dependency loop will never happen during the gossip process, but it limits the gossip possibilities for parallel workers and is not a real reflection for the original version of the gossip algorithm. Since the gossip partner needs to be in different sets, it restricts the choices of partners for gradient averaging among parallel workers. According to our experiments, although the training speed is faster than using the all-reduce method, the accuracy of the final solution decreases. Detailed experimental results are shown in section 4.4. Another potential defect of this implementation is the limited scalability. Since it needs to allocate a shared variable for each passive worker to store gossip pairing information, the number of needed shared variables will increase with the number of parallel workers.

Since the shared variables need a lock mechanism to maintain the correctness, we also consider if it will cause competition for accessing the resource. Theoretically, this lock mechanism adds to the waiting time for processes. According to our experiments, we find out that in most scenarios, this communication overhead is minor compared with the speed up achieved by using asynchronous gradient averaging.

To improve the training speed up with stable accuracy, we propose a novel way to implement asynchronous GPU parallel training. The new proposal (GossipGCN) guarantees the correctness of the algorithm by following the original version of gossip, which means each parallel worker is able to choose a gossip partner if it's not paired yet. Compared with implementation version 1, the new implementation gives more choices for the parallel workers when picking gossip partners by getting rid of the active and passive sets division. Detailed implementation of the proposed algorithm is elaborated in the following section.

4.3 Asynchronous GPU Parallel Implementation Version 2 (GossipGCN)

In this section, we propose a second version of asynchronous decentralized data parallel GCN based on graph partition method (GossipGCN). The essential idea for this implementation is similar to the version 1. We remove the synchronization barrier at the end of iterations by using the gossip algorithm to average gradients. But different from version 1, we get rid of the division of passive and active sets. Parallel workers can choose any available worker as the gossip partner. Details of the algorithm are in the following.

4.3.1 GossipGCN on a GPU cluster

The following are the main steps of our proposed algorithm, GossipGCN. Step 1(Graph partition): the original input graph is partitioned into n number of subgraphs, which are used as sample mini batches. Step 2 (Initialize work-pool): initialize a central work-pool queue containing the subgraph ids, repeated for the total number of epochs fixed a priori. Step 3 (Initialization): Initialize shared variables to keep track of gossip partner information. Each worker (GPU) initiates a local model, defines loss function and other hyperparameters. Step 4 (Gradient calculation): If the work-pool is not empty, each worker picks a work (mini batch) from the work-pool and calculates local gradients. Otherwise, go to step 7. Step 5 (Gradient averaging): Check if the averaging interval is met, if not, directly go to step 6. Otherwise, each worker checks if some gossip partner is waiting to average gradients with the current worker. If so, it averages gradients with the partner. If no partner is waiting, it randomly picks a partner from the list of available workers. Step 6 (Update): Each worker updates its local parameters and repeats from Step 4. Step 7 (Clean up): Check if any gossip-partner is waiting to average gradients with the current worker (legacy averaging call). If so, average with the partner using the most recent gradients. Step 8 (Finalization): The master waits until all workers finish training, then it compares local solutions of all workers and chooses the best one as the final solution.

More detailed pseudocode of GossipGCN is provided in the Algorithm 6 below.

Algorithm 6: GossipGCN on a GPU cluster

```
Input: Graph G, feature X, label Y
Output: GCN model with trained weights
    /* Master does the following from lines 1-8 */
    1. Partition input graph into n subgraphs G_0, G_1, ... G_n-1. The subgraphs are the mini
       batches
    2. Set epoch number as e, a pre-set value determined empirically
    3. Initialize work_pool = [0,1, ... n-1, 0, 1, ... n-1, ...  0, 1, ... n-1] /* The shared
       work_pool queue stores the mini batch indexes. Each index is stored e times */
    4. Initialize gossip_partner = [-1, -1, ...  -1, -1] /* Array gossip_partner stores gossip
       paired workers, -1 represents not paired. The size of the array is p, the number of worker
       processes. */
```

```
5.  Initialize available_workers = [0, 1, 2, ... p-1] /* Array available_workers stores
    available worker ids. Initially all workers are available */
6.  Start p worker processes, each GPU is mapped with 1 worker
7.  Initialize model on each GPU
8.  Initialize averaging_interval. /* The averaging_interval determines how many mini
    batches are processed by a worker before averaging gradients with a gossip partner */
    /* Each worker does the following from lines 9-54 */
9.  Initialize count = 0
10. Initialize partner = -1
11. while work_pool is not empty do
12.     Remove a mini batch index i from work_pool
13.     Increment count by 1
14.     Get G_i features and labels from CPU
15.     Calculate local gradients using G_i
16.     if count is multiple of averaging_interval then
17.         Lock available_workers.
18.         Lock gossip_partner
19.         Set my_partner = gossip_partner[proc_id] /* proc_id is process id of worker*/
20.         if my_partner == -1 then
21.             Remove proc_id from available_workers
22.             if available_workers is not empty then
23.                 partner = randomly choose an id from available_workers
24.                 remove partner from available_workers
25.                 gossip_partner[proc_id] = partner
26.                 gossip_partner [partner] = proc_id
27.             else
28.                 Add proc_id to available_workers
29.         else
30.             partner = my_partner
31.         Unlock available_workers
32.         Unlock gossip_partner
33.         if partner > -1 then
34.             /* Introduce default barrier among paired-up processes */
35.             Average gradients with partner
```

```
36.             Lock *available_workers*
37.             Lock *gossip_partner*
38.             Reset *gossip_partner[proc_id] = -1*
39.             Add *proc_id* to *available_workers*
40.             Unlock *available_workers*
41.             Unlock *gossip_partner*
42.             Reset *partner = -1*
43.         Update weights for local model
44. **end while**
    /* Clean-up steps */
45. Lock *available_workers*
46. Lock *gossip_partner*
47. **if** *proc_id* is in *available_workers* **then**
48.     Remove *proc_id* from *available_workers*
49. **else**
50.     *my_partner = gossip_partner[proc_id]*
51.     Average gradients with *my_partner* using the gradients in last iteration
52.     Reset *gossip_partner[proc_id] = -1*
53. Unlock *available_workers*
54. Unlock *gossip_partner*
    /* Master chooses the final solution */
55. Compare the accuracies of models on each GPU, select the best one as final model
```

In the previous, the value of *averaging_interval* is determined empirically, and it sets a compromise between performance and accuracy, i.e. more frequent pairing with a gossip partner could give a more accurate model however at the cost of performance. Figure 4.2 shows an example scenario of how the values of *gossip_partner* and *available_workers* are changed when 4 GPUs are used for training. Note that any failed pairing (i.e., a worker who could not pair up with a gossip partner for various reasons) is handled during the clean-up steps (lines 45-54 in Algorithm 2).

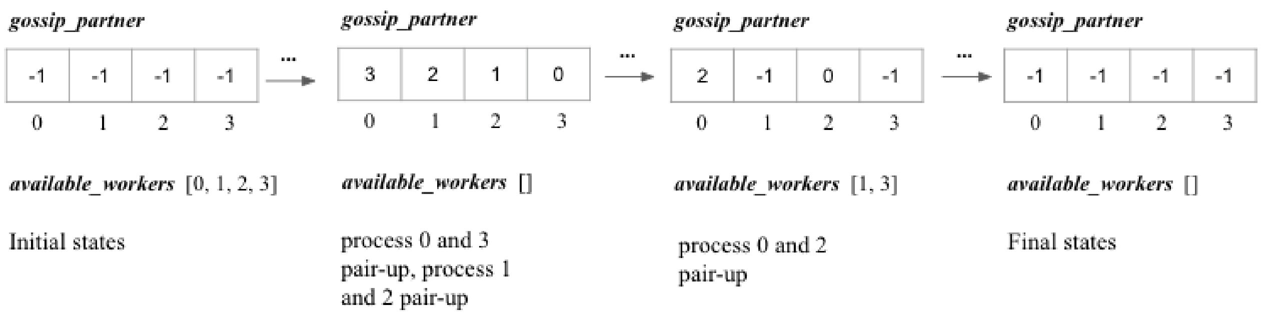


Figure 4.2 An example run using 4 GPUs

As a major difference with the synchronous all-reduce in Algorithm 3 and 4, here only two of the workers (i.e. gossip partners) need to synchronize with each other to exchange their local gradients. So, a straggler can affect only its partner as compared to all the other workers in a synchronous approach as illustrated in Figure 4.3. Moreover, dynamic work loading by using the work-pool can greatly reduce any load imbalances among GPUs and hence significantly enhance efficiency.

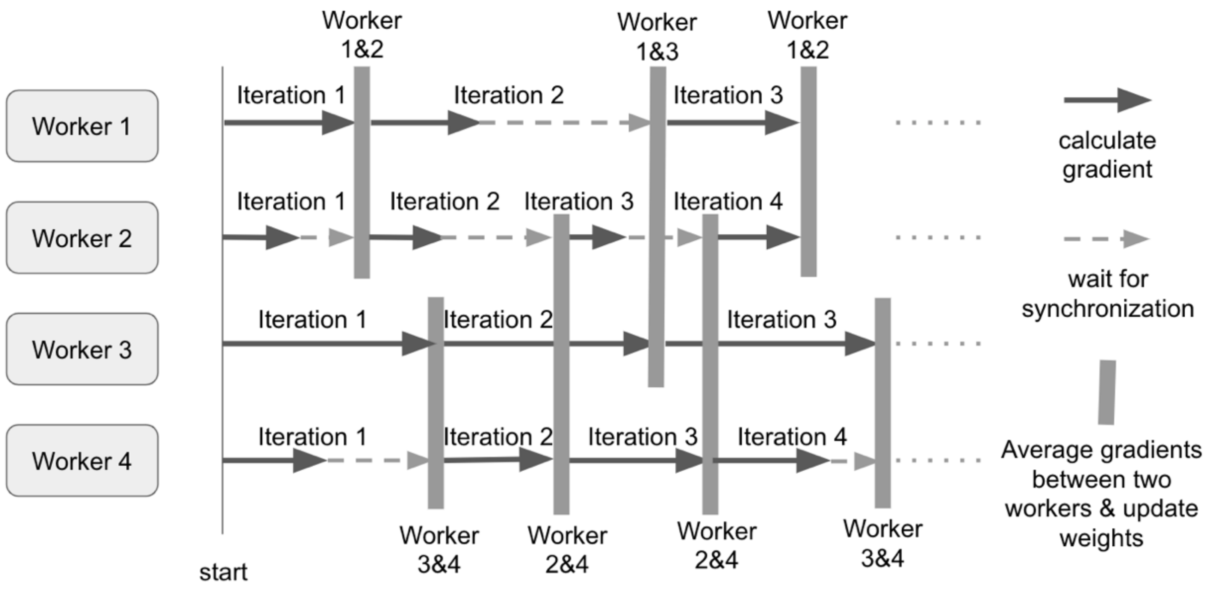


Figure 4.3 Gradient averaging scheme for GossipGCN.

In the synchronous algorithm, all workers get the same number of mini batches for the training. However, in the asynchronous method using gossip algorithm, parallel workers may have different numbers of iterations according to their training speed. Fast workers can get more mini batches than the slower ones. The advantage for GossipGCN is that it maximizes the usage of calculation capability for fast workers, since it allows fast GPUs to have more workload. But this may lead to a potential problem of decreased accuracy, because the gradients calculated in two workers are based on different numbers of iterations. This problem is known as gradient staleness. The shuffling of synchronization workers can help to mitigate this problem.

The previous algorithm has been implemented on a GPU cluster using PyTorch as the deep learning framework. PyTorch has become an important tool for GCN training and has gained popularity in recent years due to its robust support for GPUs. PyTorch supports implementation of neural networks on GPU using CUDA extensions and facilitates GCN implementation with the help of PyG (PyTorch Geometric Library) [26]. In addition, PyTorch integrated NCCL backend can perform direct GPU-to-GPU communication, thus bypassing the CPU. This makes it possible for direct and efficient gossip-pairing and communication among GPUs, which could not be possible in the past. More about implementation details and experimental results are presented in Section 4.4.

Figure 4.4 illustrates the architecture of the proposed algorithm GossipGCN and shows how the gossiping averaging gradient is achieved using the shared variables. The workers first request an id from the *work_pool* and calculate gradients locally. For *p* number of parallel workers, the shared variable *gossip_partner* is initialized with a list of size *p* filled by "-1", which means there is no neighbor to synchronize at the beginning. The index of this list represents process id, and the value at index *i* represents the gossip neighbor id for process *i*. The other shared variable *available_workers* represents available workers for gradient averaging and is initialized with all processes' ids. When local gradients are ready, each worker gets the values in *gossip_partner* and *available_workers*.

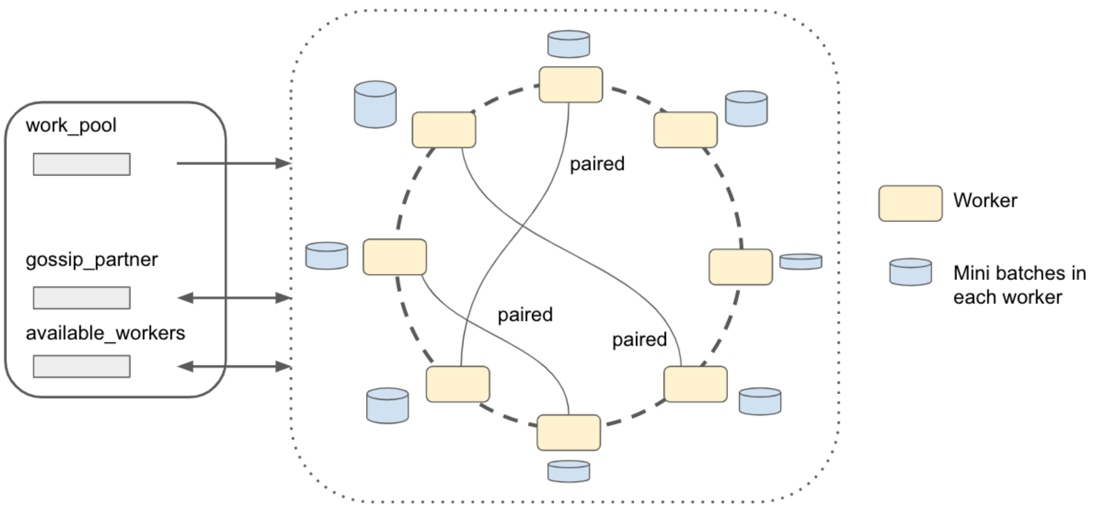


Figure 4.4 Gossip scheme with work-pool.

Since each process is mapped with one GPU, the process id can represent the GPU id that calculates the local gradients. The gossip partner id of the current process *my_sync* can be obtained at the index of the current process id by calling *gossip_partner[proc_id]*. There are two cases for the value of *my_sync*. In one case, if the value of *my_sync* is -1, it means the current process has no workers waiting for pair-up, so it can pick an id from *available_workers*. After picking the gossip neighbor, *available_workers* and *gossip_partner* will be updated respectively. The value of *my_sync* is updated to the selected neighbor's process id. And the two processes' ids are removed from *available_workers*, indicating that the two workers are already paired and cannot be chosen by a third worker to synchronize. In the other case, if the value of *my_sync* is not -1, it means there is another process waiting to average gradients with the current process.

After the value of *my_sync* is determined, which means the gossip partner is known. The two GPUs will average their gradients and update weights with the same gradients. When the update is done, each process can put the process id back to *available_workers* and reset the corresponding value in the *gossip_partner*. This finishes the iteration, and the workers are ready to ask for next mini batches from the *work_pool* until it becomes empty.

After the training phase, we implement a clean-up step to avoid program. Because it is possible that when the work-pool is empty, a worker already puts the current process id back to *available_workers*, and another worker asks to average gradients with it. So, in the clean-up step, each worker checks if there is any remaining gradient averaging request. If the value at *gossip_partner[proc_id]* is not -1, it means there is a worker waiting to average gradients. Then, the current worker will use the old gradients in the last iteration to perform gradient averaging and safely end the process.

Since the spare computing capacity of each GPU can be very different, it is possible that different workers get various numbers of mini batches, which means the local iteration numbers are different. If GPUs work at extremely different pace, the slower worker may not have enough iterations to reach a desired accuracy. Although exchanging parameters with other workers can help to gather more information, it may still result in a low accuracy. On the other hand, the fast worker can get more mini batches to train the local model and can reach a better accuracy. To get the optimal solution, we compare the accuracies of all local models in parallel workers and choose the best one as the final optimal solution.

4.3.2 Work-pool

As discussed in Chapter 3, one of the disadvantages for synchronous AllReduce-GCN is that the mini batch numbers are fixed and divided evenly for all workers. This is not a good methodology since it wastes the computational capacity on fast workers and leads to a large idling time. The synchronization barrier at the end of the iterations prevents the AllReduce-GCN to adapt a dynamic loading mechanism. This limitation is discussed detailly in the previous chapter.

To improve the speed up and reduce time wasted on waiting idly, we implemented a work-pool mechanism to make the best use of the computing capacity of each worker. In our proposed method (GossipGCN) we create a shared queue to store the ids of mini batches. Each worker can request a mini batch from the work-pool whenever it is free. In addition, we modify the traditional two-level *for* loop in deep learning to using a single level *while* loop. This improves the parallel level for the algorithm and is illustrated in Algorithm 6.

In such an implementation, we first use the same random partition method as in AllReduce -GCN to get *n* subgraphs:

$$G = [G\_0, G\_1, \ldots G\_n-1].$$

Since each subgraph is a mini batch, the index queue *I* for the mini batches is:

$$I = [0, 1, 2, \ldots n-1].$$

We define epoch number is *e*, so, the total work amount for all epoch can be represented as the mini batch index queue:

$$work\_pool = [\underbrace{0,1, \ldots n-1,}_{epoch\ 1} \underbrace{0, 1, \ldots n-1,}_{epoch\ 2} \ldots \underbrace{0, 1, \ldots n-1}_{epoch\ e}]$$

The shared variable *work_pool* is accessible to all parallel workers. During the training, each worker requests an index from *work_pool* and sends the corresponding mini batch to the GPU for calculating stochastic gradients. After the current iteration is finished, a worker can request another mini batch index from *work_pool* until it is empty.

The correctness of the algorithm is maintained by internally implemented locking semantics of the multiprocessing.Queue module in PyTorch. It blocks competitive requests and allows getting the index one by one. Using this mechanism makes sure the same amount of data is trained as using the all-reduce method, and it helps to reduce idling time for GPUs since a faster GPU is possible to do more work.

4.3.3 Optimization with Periodic Gradient Averaging

In this section, we discuss the periodic gradient averaging method used in GossipGCN Although using gossip method can reduce the idling time for fast GPUs, gradient transmission and waiting for a slow gossip partner still cause large communication overhead. Similar to AllReduce-GCN (Algorithm 4), the periodic gradient averaging is used to reduce GPU communication frequency. This method helps push the speed up limit for parallel GCN training further. Figure 4.5 shows the scheme of GossipGCN with periodic gradient averaging.

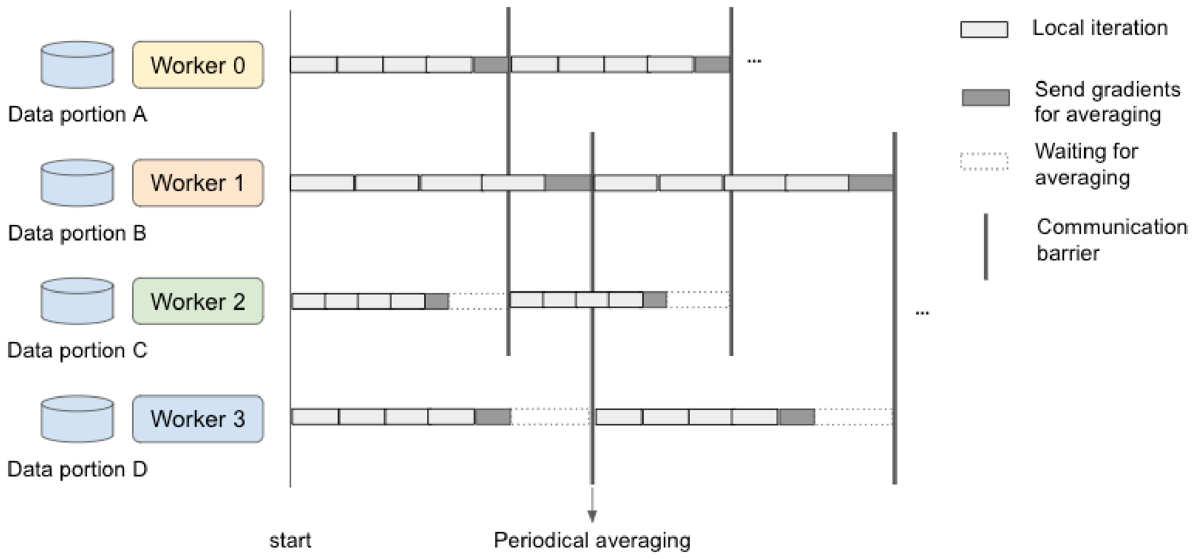


Figure 4.5 GossipGCN with periodic gradient averaging.

In Algorithm 6, firstly, a gradient averaging interval (*averaging_interval)* is set before the training starts. Then during the training process, each worker keeps a variable *count* to track the number of mini batches trained locally. Workers request a mini batch id from the work-pool in each iteration and calculate gradients respectively. The value of *count* is initialized to 0 and is incremented by 1 after a mini batch is taken. If *count* is a multiple of *averaging_interval*, it means the gradient averaging interval is reached and the worker will execute the averaging. The synchronization interval is chosen empirically and can be different for various datasets.

It is mentioned previously that asynchronous GCN training using gossip method can cause iteration gaps among parallel workers. Although using periodic gradient averaging allows fast workers to make full usage of the calculation capacity, the iteration gap between the slowest worker and the fastest worker is enlarged at the same time. So, there is a trade-off between accuracy and training speed. Even though continuous gossiping with random workers can help to reduce the effect of iteration gaps, if the gap is too large, it may cause huge loss in accuracy. So, it is important to keep the iteration gap within a safe threshold. We have tested our algorithm using various gradient averaging intervals. The experimental results are elaborated in section 4.4.

4.3.4 Theoretical Discussion

In this section, we present proofs and theoretical analysis for the proposed algorithm GossipGCN. We prove that GossipGCN is deadlock-free and guarantees the training can be finished properly when the program ends. In the following discussion, the assumptions are that hardware, software, and network are fault free and hence processes do not crash or wait indefinitely due to external factors.

Lemma 1 (Deadlock): GossipGCN is deadlock-free.

Proof: All workers acquire the locks of the shared arrays, *available_workers* and *gossip_partner* in the same order. This guarantees that there cannot be a circular wait in acquiring of the locks and hence no deadlock is possible.

Next, we show that no deadlock is possible in pairing-up with a gossip partner (lines 19-30 in Algorithm 6). We prove this by contradiction. Assume that there exists a deadlock in the gossip-pairing. Then there must be a circular wait involving at least 3 workers, say workers A, B, and C such that worker A is waiting to exchange gradients with worker B, worker B is waiting to exchange gradients with worker C, and worker C is waiting to exchange gradients with worker A. Suppose worker A is the first one to grab both the locks of available_workers and gossip_partner. In that case, worker A will update its pairing information with worker B in gossip_partner (lines 25-26) and also remove itself and worker B from available_workers (lines 21 and 24). So, subsequently when worker B grabs the locks, there is no possibility that worker B would choose worker C as partner because its partner is already set as worker A (line

19); similarly, when worker C grabs the locks, there is no possibility that it would choose worker A as its partner. As a result, circular wait is not possible and hence there is no deadlock. This concludes the proof.

Lemma 2 (Termination): In GossipGCN, each worker terminates without indefinitely waiting for its gossip partner to exchange information.

Proof: A worker A pairs up with its randomly selected gossip partner (worker B) if and only if the work-pool is not empty and the averaging interval is met (lines 11-30 in Algorithm 6). Subsequently, worker A waits for worker B to exchange gradients at an implicit barrier for exchanging information (line 35). There are two scenarios where worker B will meet the barrier and exchange information with worker A as follows. Case 1: worker B meets the barrier at line 35 and exchanges information with worker A. Case 2: worker B meets the barrier at line 51, which is reached because the work-pool is empty and the while loop (line 11) is exited by worker B. In either case, worker A and B always meet at the barrier and exchange information. As a result, none of the workers will wait indefinitely and hence will terminate normally. This concludes the proof.

Gossip algorithm is first used for consensus problems, and the goal is to exchange messages with workers until a consensus is made. Previous research has been done to prove that gossip algorithm can be used to solve averaging problems [46, 63, 64]. In such problems, each node in the network has a local value and needs to achieve a global average of the values at all nodes in the network.

One of the basic gossip methods is making randomly paired workers to average their values until it converges to the global average. This method is also called Pairwise Gossip and is proposed in [46]. Suppose a parallel worker *i* has a value x_i, in each gossip step, it needs to randomly choose a neighbor *j* and average value with it. After the averaging, the values at both workers become:

$$x_i(t+1) = x_j(t+1) = 0.5\, x_i(t) + 0.5\, x_j(t)\ .$$

The authors also suggest that the pairwise gossip method is not only suitable for averaging problems but also applicable to calculate global minimum or maximum. The convergence of gossip algorithm is proved in [63]. It shows that the consensus can be achieved given the condition that the gossip pairs are chosen equally at random. [48] provides theoretical proof for the correctness of a large set of asynchronous gossip-based algorithms. A detailed mathematical proof can be found in the original paper.

In recent years, gossip algorithm is applied to deep learning problems for calculating gradient averages among parallel workers. Theoretical analysis for asynchronous parallel training is given in previous research [14-16]. It is proved that the convergence can be achieved among all local models, and it is consistent with baseline training if the iteration number is large enough. The completed proof is provided in [15]. A strategy for proving gossip algorithm on parallel training is given in [14]. Because of gradient averaging, a lemma is taken that parallel compute nodes have the same cost function. Since shuffling of gossip pairs ensures that the training samples are considered multiple times in different workers over time, the local cost functions are optimized based on all samples. Detailed proofs and results can be found in those works mentioned above.

4.4 Experimental Evaluations

In this section, we present evaluations for the proposed algorithms using the same datasets (Reddit and Amazon dataset) as described in Chapter 3. Detailed information about the datasets is given in Table 3.1. We evaluate the performance of GossipGCN with various gradient averaging intervals, and the intervals are chosen according to empirical experience. Since synchronous all-reduce is the most popular approach for gradient averaging on GPUs, the performance of GossipGCN is compared with the all-reduce counterpart AllReduce-GCN (refer to Chapter 3) for various averaging intervals.

4.4.1 Implementation Details

In this research, since the graph partition phase is the same for different implementations, we measure the partition time separately from the real training time (calculating stochastic gradients and updating model). The averaged partition times are 1.57 seconds for Reddit dataset and 198.42 seconds for Amazon dataset. For both datasets, the hyperparameters used in experiments are the same as the ones used for synchronous AllReduce-GCN. Details of the hyperparameters can be found in Table 3.3. We implement a work-pool mechanism and a gossip algorithm to communicate gradients among GPUs. The implementation is done using PyTorch [12] and the baseline implementation is presented in the previous chapter. The backend for GPU communication is NCCL embedded in PyTorch.

The experiments are done using a single compute node (8 GPUs) as presented before. We use the averaged values of experimental data to produce the tables and figures in this section.

One challenge of the proposed implementations is to share variables among different workers. There are different ways to share data among processes, one convenient method is to use torch.multiprocessing.Queue [58]. The shared variable *work_pool* is implemented using multiprocessing.Queue in PyTorch. The torch.multiprocessing module is a replacement for the multiprocessing module of Python [65] and shares the same operations. The data sent through multiprocessing.Queue (mp.Queue) is moved to shared memory and other processes can access it through a handler. Since there is a speed limitation for the pipe, mp.Queue is fast for light weight data but slow for large data such as a subgraph. So, we store the mini batch ids instead of the original subgraphs in the shared queue.

The mp.Queue class implements locking mechanisms internally to maintain correctness. It temporarily blocks competing requests, for example, when two processes ask for mini batch ids at the same time. Theoretically, the locking semantics may cause bottlenecks. But, since it is very fast for light weight data (e.g. small integers), the time spent on waiting for locks can be ignored. Our experiments prove that the locking mechanism has little effect on the speed up of parallel-GPU training.

Since the mp.Queue class is fast for exchanging information among processes, we create two shared variables (*available_workers* and *gossip_partner*) using mp.Queue to maintain the correctness of the gossip algorithm and avoid deadlocks. This implementation not only guarantees the correctness of the gossip algorithm but also avoids the high cost of sending messages for broadcasting neighbor ids.

Another challenge of the implementation is one-to-one communication between GPUs using PyTorch. Since local model and stochastic gradients are stored in GPU, we use embedded NCCL backends in PyTorch to perform gradient averaging. Since PyTorch does not support NCCL point-to-point communication for now, we create subgroups for the paired processes using torch.distributed.new_group and call torch.distributed.all_reduce within the subgroups to mimic point-to-point communication.

4.4.2 Preliminary Experimental Results for Implementation Version 1

In the preliminary experiments for the asynchronous GPU parallel implementation version 1, we test parallel training on 4 GPUs for the Reddit dataset. We observe that although the training speed is faster than AllReduce-GCN, there is an accuracy loss for this implementation. The accuracy also drops when the averaging interval grows as shown in Table 4.1. The main reason for this accuracy loss is that the fixed division for active and passive workers limits the choices of gossip partners. Since workers cannot average gradients with other workers in the same set, it reduces the chance to get complete information from mini batches that are trained in parallel workers.

Table 4.1 Asynchronous GPU parallel implementation version 1 for Reddit dataset (4 GPUs)

Averaging_interval	1	20	100
Time / sec	13.19	10.28	9.77
F1 score	0.9532	0.9514	0.9513

It is also observed in the experiments that the performance of this implementation is not stable for a system with stragglers. When slow workers reside in the active set and fast workers reside in the passive set, the gradients may be averaged only a few times, especially when using periodic averaging. As we use a work-pool to allocate jobs, fast workers can get more mini batches than slow workers. However, workers in the passive set cannot pick another worker to average gradients. The total number of gradient averaging times are decided by the number of mini batches trained by workers in the active set. If these workers are slow, then the total number of gradient averaging times are small. This results in a low accuracy because a large portion of mini batches are only used for local model update.

In a real-world scenario, the workloads in GPUs may be very different. Thus, GPUs of the same make can work at different paces. According to our experiments, the gossip implementation version 1 is not a superior solution for asynchronous GPU parallel training. We propose a second version for the gossip training implementation (GossipGCN), and the experimental results are discussed in the following section.

4.4.3 Experimental Results for GossipGCN

Our experiments show that in most scenarios, the proposed GossipGCN can achieve better performance compared to the synchronous AllReduce-GCN. Tables 4.2 ~ 4.5 illustrate the parallel training times and accuracies of GossipGCN for Reddit and Amazon dataset using different numbers of GPUs. Tables 4.6 and 4.7 elaborate the speed up comparisons for AllReduce-GCN and GossipGCN for the two datasets. Tables 4.8 and 4.9 show comparisons of accuracy and locally trained mini batch numbers for AllReduce-GCN and GossipGCN using 8 GPUs.

As the results illustrate, GossipGCN outperforms AllReduce-GCN in training speed. It is also noticeable that the training finishes faster with a larger averaging interval, which can be attributed to a smaller communication overhead with a larger averaging interval. The results also illustrate that the work-pool strategy in GossipGCN balances workload among different GPUs which is unlike the fixed workload in AllReduce-GCN.

Table 4.2 GossipGCN for Reddit dataset (4 GPUs)

Averaging_interval	1	20	100
Time / sec	12.46	11.74	10.62
F1 score	0.9573	0.9599	0.9613

Table 4.3 GossipGCN for Reddit dataset (8 GPUs)

Averaging_interval	1	20	100
Time / sec	9.62	7.44	5.68
F1 score	0.9502	0.9566	0.9538

Table 4.4 GossipGCN for Amazon dataset (4 GPUs)

Averaging_interval	1	20	100
Time / sec	73.80	56.81	51.97
F1 score	0.7537	0.7545	0.7518

Table 4.5 GossipGCN for Amazon dataset (8 GPUs)

Averaging_interval	1	20	100
Time / sec	61.65	41.39	29.42
F1 score	0.7222	0.7393	0.7401

Table 4.6 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit dataset.

GPU number		1	4	8
AllReduce-GCN Speed up	averaging_interval = 1	-	1.87	2.93
	averaging_interval = 20	-	2.13	2.96
	averaging_interval = 100	-	2.19	2.97
GossipGCN Speed up	averaging_interval = 1	-	2.40	3.11
	averaging_interval = 20	-	2.55	4.02
	averaging_interval = 100	-	2.82	5.27

Table 4.7 Speed up comparison of AllReduce-GCN and GossipGCN for Amazon dataset.

GPU number		1	4	8
AllReduce-GCN Speed up	averaging_interval = 1	-	1.74	2.81
	averaging_interval = 20	-	2.07	3.05
	averaging_interval = 100	-	2.27	3.39
GossipGCN Speed up	averaging_interval = 1	-	2.44	2.93
	averaging_interval = 20	-	3.17	4.36
	averaging_interval = 100	-	3.47	6.13

Table 4.8 Comparison of parallel training for Reddit Dataset (8 GPUs)

Algorithm	AllReduce			Gossip		
	averaging_ interval = 1	averaging_ interval = 20	averaging_ interval = 100	averaging_ interval = 1	averaging_ interval = 20	averaging_ interval = 100
Time (in seconds)	10.21	10.11	10.07	9.62	7.44	5.68
F1-score	0.9609	0.9449	0.9435	0.9502	0.9566	0.9538
Number of mini batches in each GPU	150 | 150 | 150 | 150 |150 | 150 | 150 | 150			130 | 127 | 188 | 204 | 119 | 162 | 123 | 147	122 | 123 | 215 | 264 | 108 |173 | 102 | 93	108 | 91 | 92 | 301 | 134 | 227 | 143 |104

Table 4.9 Comparison of parallel training for Amazon Dataset (8 GPUs)

Algorithm	AllReduce			Gossip		
	averaging_ interval = 1	averaging_ interval = 20	averaging_ interval = 100	averaging_ interval = 1	averaging_ interval = 20	averaging_ interval = 100
Time (in seconds)	64.18	59.13	53.20	61.65	41.39	29.42
F1-score	0.7721	0.7189	0.7187	0.7222	0.7393	0.7401
Number of mini batches in each GPU	2400 | 2400 | 2400 | 2400 | 2400 | 2400 | 2400 | 2400			2237 | 2816 | 2105 | 2214 | 2386 | 2761 | 2552 | 2129	2310 | 2483 | 2905 | 2831 | 2014 | 1936 | 2673 | 2048	2599 | 2316 | 3185 | 2954 | 2006 | 1728 | 2864 | 1548

It can be observed from the experiments that parallel GCN training suffers accuracy loss, irrespective of whether it is based on all-reduce or Gossip, as compared to the baseline training on single GPU. The reason for accuracy loss in parallel training is that information of multiple mini batches is gathered to make one update of the model; while in the baseline training on single GPU one mini batch is trained at one step of model update; hence successive training of mini batches is done with updated models. This can affect the accuracy of the final output model [66]. For GossipGCN, it experiences accuracy loss because of the same reason for parallel training. Moreover, since different workers may train various numbers of subgraphs when using GossipGCN, the iteration gap can lead to lower accuracy.

However, it can be seen that GossipGCN has smaller accuracy loss with higher averaging interval as compared to AllReduce-GCN. Intuitively, lower averaging frequency can cause loss in accuracy. In AllReduce-GCN, accuracy loss is affected greatly by reducing averaging frequency, since some information may be lost in the training. However, in GossipGCN, the algorithm is more flexible in requesting jobs, and the fast worker can train more subgraphs locally. The parameters of the local model in the fast GPU are updated based on more information of the mini batches, thus the accuracy of the output model is higher than the AllReduce-GCN. To minimize the effect of accuracy loss, we compare the local models on each GPU and choose the best one as the final output model.

Figures 4.6 ~ 4.8 exhibit the speed up comparison for AllReduce-GCN and GossipGCN using different averaging intervals. From the figures we can see that by reducing gradient averaging frequency, GossipGCN can achieve a higher speed up compared with AllReduce-GCN. By comparing the results, we observe that using gossip averaging method can push the speed up limit further with periodic gradient averaging.

It is also observed from experiments that there is a rare scenario when all GPUs are totally free. This only happens when there is no other job running on the GPUs. In this scenario, the training can be done very fast. Since all calculations and gradient averaging are done immediately and no workers wait idlily, changing averaging intervals has little influence on the total training time. However, this situation happens very occasionally, in most scenarios, performing periodic gradient averaging can help to improve the training speed.

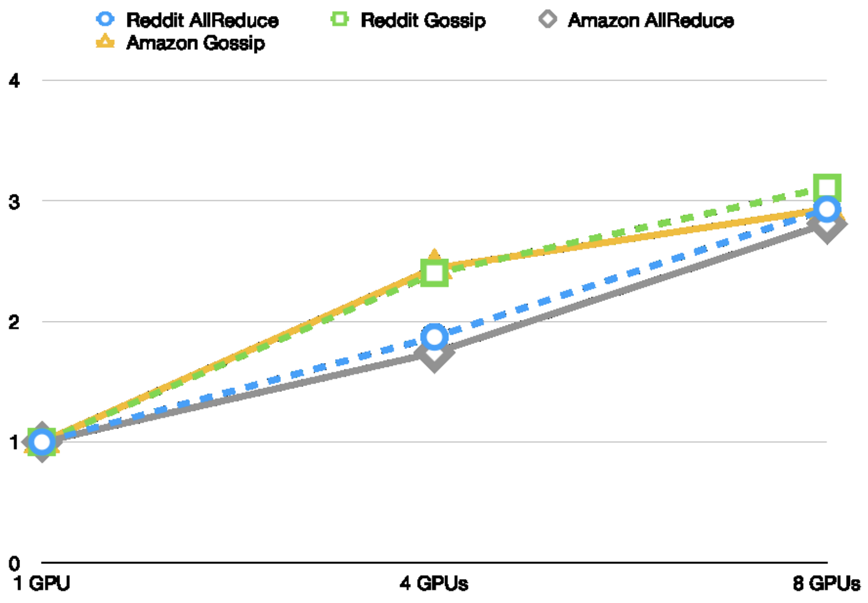


Figure 4.6 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit and Amazon dataset with averaging_interval = 1.

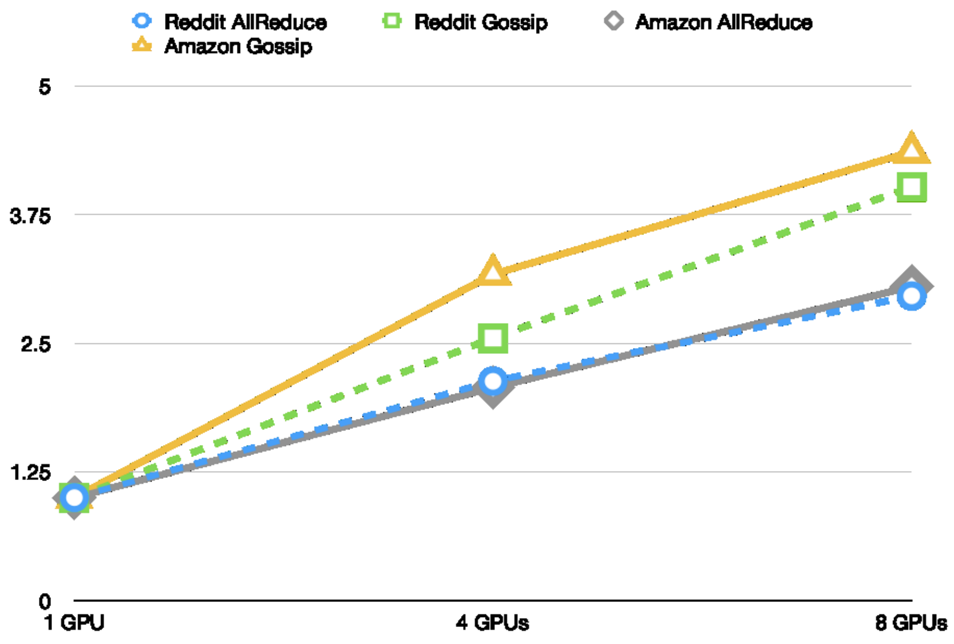


Figure 4.7 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit and Amazon dataset with averaging_interval = 20.

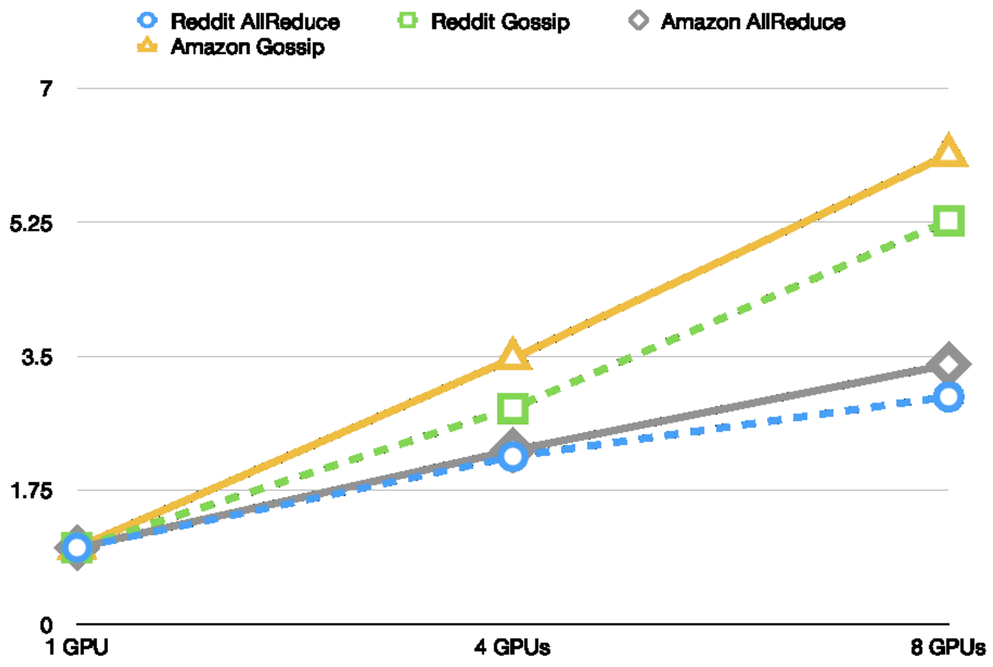


Figure 4.8 Speed up comparison of AllReduce-GCN and GossipGCN for Reddit and Amazon dataset with averaging_interval = 100.

We can see from experimental results that there is a trade-off for the training speed and accuracy. Taking both training time and accuracy into consideration, performing gradient averaging in each iteration is not the best choice. GossipGCN with periodic synchronization increases the speed up largely with minor accuracy loss on a real-world GPU cluster where the status of each GPU changes fast. For different datasets and systems, the averaging interval can be adjusted in experiments to get the superior performance.

Another issue we notice is that multiple NCCL calls will compete for the communication bandwidth among GPUs. If there are multiple jobs requesting communication between GPUs using all-reduce or other collective calls, the training time may be affected. AllReduce-GCN has only one NCCL call (the all-reduce call) at one time, while in GossipGCN, it is possible that multiple NCCL calls happen at the same time when more than two sets of paired workers start to average gradients.

According to the experimental results, we see that in most scenarios, using GossipGCN achieves better speed up than AllReduce-GCN, since GossipGCN eliminates the synchronization barrier for all parallel workers and only needs to average gradient with one neighbor. In a rare scenario, the GPUs have enough calculation capability to run all submitted jobs simultaneously, and there is only one collective call such as all-reduce among different GPUs in one compute node. Since the problem of synchronization barrier is not obvious in this case and multiple collective calls in GossipGCN slow down the training speed, AllReduce-GCN has better speed up than GossipGCN. However, this scenario happens very occasionally, and it is almost impossible to maintain. In most scenarios, even though multiple GPU collective calls cause more overheads in GossipGCN, the gains of reducing idling time is greater than the overheads. Generally speaking, GossipGCN has better performance than AllReduce-GCN.

4.5 Summary

In this chapter, we first provide a preliminary implementation of asynchronous and decentralized approach for GCN deep learning on a GPU cluster using the active and passive sets of workers. Then, to achieve better performance, we present a second version of the asynchronous GCN training using gossip algorithm to exchange stochastic gradients (GossipGCN). We also implement a work pool mechanism to balance workloads among workers. It is proved that the proposed algorithm is deadlock-free. Experiments show that GossipGCN outperforms AllReduce-GCN in most scenarios with periodic gradient averaging.

Generally, it achieves better speed up for graph-partition based GCN training with a stable accuracy. The algorithm is superior to the synchronous counterpart, especially in a non-dedicated system, where the workload on each GPU varies and keeps changing with time.

Chapter 5 Conclusion and Future Works

Graph convolutional networks (GCNs) play an important role in deep learning for graph related data and are widely used in many disciplines. Our research is based on a novel GCN algorithm (Cluster-GCN) that uses graph partition method instead of traditional sampling method to get mini batches.

In this thesis, we present an asynchronous and decentralized algorithm (GossipGCN) for GCN deep learning on a GPU cluster. We implement a work-pool mechanism and a gossip algorithm for GPUs to average stochastic gradients. Our method is especially suitable for a dynamic training system, where the workload on each GPU varies and keeps changing with time. In addition, inspired by local SGD and model averaging, we explore how gradient averaging frequency can affect training speed and accuracy.

In the synchronous GPU-parallel implementation (AllReduce-GCN), reducing gradient averaging frequency can help to accelerate the overall training when the gradient size is large. However, in a system with stragglers, there is a limit for the speed up because of the all-reduce barrier for gradient averaging. In GossipGCN, we use Gossip to remove the synchronization barrier. It only requires gradient averaging among two workers. In addition, the work-pool mechanism helps to balance the workload for all workers.

We implement the algorithms using PyTorch [58] and DGL [27] as the deep learning platform and libraries. The experiments are carried out using datasets of different sizes (Reddit and Amazon datasets). Generally, results show that GossipGCN achieves a better speed up and a more stable accuracy than the traditional synchronous counterpart on a real-world GPU cluster in most scenarios, especially when the GPUs are busy with multiple jobs and calculate stochastic gradients in different paces.

In the future, we suggest conducting further research on comparison of partition methods to better understand if different partition methods have an effect on training time and accuracy. Additionally, we only concentrate on the parallelizing training phase in our research, and there exist various parallel graph partitioning methods (e.g. [67]). It will help to achieve better overall GCN training speed up if we can partition the input graph and prepare mini batches in a parallel fashion. Moreover, our research focuses on asynchronous training within a single compute node. In the future, if we have access to more resources, we suggest carrying out further research to integrate the proposed method with existing asynchronous training methods in a distributed system with multiple compute nodes. Finally, the proposed algorithm can also be modified to adapt with different variants of GCNs.

Reference

[1] J. Qiu, J. Tang, H. Ma, Y. Dong, K. Wang, and J. Tang, “DeepInf: Social Influence Prediction with Deep Learning,” in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, London United Kingdom, Jul. 2018, pp. 2110–2119, doi: 10.1145/3219819.3220077.

[2] M. Zhang and Y. Chen, “Link Prediction Based on Graph Neural Networks,” in Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 5171–5181.

[3] E. Choi et al., “Learning the Graphical Structure of Electronic Health Records with Graph Convolutional Transformer,” in Proceedings of the AAAI Conference on Artificial Intelligence, 2020, vol. 34, pp. 606--613.

[4] S. Rhee, S. Seo, and S. Kim, “Hybrid Approach of Relation Network and Localized Graph Convolutional Filtering for Breast Cancer Subtype Classification,” in Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, 2018, pp. 3527–3534, doi: 10.24963/ijcai.2018/490.

[5] D. Nathani, J. Chauhan, C. Sharma, and M. Kaul, “Learning Attention-based Embeddings for Relation Prediction in Knowledge Graphs,” in Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, 2019, pp. 4710–4723.

[6] J. Zhou et al., “Graph Neural Networks: A Review of Methods and Applications,” ArXiv181208434 Cs Stat, Jul. 2019, [Online]. Available: http://arxiv.org/abs/1812.08434.

[7] M. Gori, G. Monfardini, and F. Scarselli, “A new model for learning in graph domains,” in Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., Montreal, Que., Canada, 2005, vol. 2, pp. 729–734, doi: 10.1109/IJCNN.2005.1555942.

[8] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, “Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks,” in Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, Anchorage AK USA, Jul. 2019, pp. 257–266, doi: 10.1145/3292500.3330925.

[9] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, “GraphSAINT: Graph Sampling Based Inductive Learning Method,” presented at the International Conference on Learning Representations, 2019.

[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud,” Proc. VLDB Endow., vol. 5, no. 8, pp. 716–727, Apr. 2012, doi: 10.14778/2212351.2212354.

[11] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “Powergraph: Distributed graph-parallel computation on natural graphs.,” in 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12), 2012, pp. 17–30.

[12] “PyTorch DistributedDataParallel.” 2020, [Online]. Available: https://pytorch.org/docs/stable/nn.html#torch.nn.parallel.DistributedDataParallel.

[13] S. Pal et al., “Optimizing Multi-GPU Parallelization Strategies for Deep Learning Training,” IEEE Micro, vol. 39, no. 5, pp. 91–101, Sep. 2019, doi: 10.1109/MM.2019.2935967.

[14] J. Daily, A. Vishnu, C. Siegel, T. Warfel, and V. Amatya, “GossipGraD: Scalable Deep Learning using Gossip Communication based Asynchronous Gradient Descent,” CoRR, vol. abs/1803.05880, 2018, [Online]. Available: http://arxiv.org/abs/1803.05880.

[15] X. Lian, W. Zhang, C. Zhang, and J. Liu, “Asynchronous Decentralized Parallel Stochastic Gradient Descent,” in Proceedings of the 35th International Conference on Machine Learning, 2018, vol. 80, pp. 3043--3052, [Online]. Available: http://proceedings.mlr.press/v80/lian18a.html.

[16] M. Assran, N. Loizou, N. Ballas, and M. Rabbat, “Stochastic Gradient Push for Distributed Deep Learning,” in International Conference on Machine Learning, 2019, pp. 344--353.

[17] “NVIDIA Collective Communications Library (NCCL).” 2020, [Online]. Available: https://developer.nvidia.com/nccl.

[18] J. Hu, C. Guo, B. Yang, and C. S. Jensen, “Stochastic Weight Completion for Road Networks Using Graph Convolutional Networks,” in 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, Macao, Apr. 2019, pp. 1274–1285, doi: 10.1109/ICDE.2019.00116.

[19] X. Geng et al., “Spatiotemporal Multi-Graph Convolution Network for Ride-Hailing Demand Forecasting,” Proc. AAAI Conf. Artif. Intell., vol. 33, pp. 3656–3663, Jul. 2019, doi: 10.1609/aaai.v33i01.33013656.

[20] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, “A Comprehensive Survey on Graph Neural Networks,” IEEE Trans. Neural Netw. Learn. Syst., pp. 1–21, 2020, doi: 10.1109/TNNLS.2020.2978386.

[21] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, “Benchmarking Graph Neural Networks,” ArXiv200300982 Cs Stat, Jul. 2020, Accessed: Nov. 27, 2020. [Online]. Available: http://arxiv.org/abs/2003.00982.

[22] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” presented at the International Conference on Learning Representations (ICLR 2017), 2017.

[23] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive Representation Learning on Large Graphs,” in Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 1025–1035.

[24] J. Chen, T. Ma, and C. Xiao, “FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling,” presented at the International Conference on Learning Representations, 2018.

[25] J. Chen, J. Zhu, and L. Song, “Stochastic Training of Graph Convolutional Networks with Variance Reduction,” in Proceedings of the International Conference on Machine Learning, 2018, pp. 942–950.

[26] “PyTorch Geometric Library (PyG).” 2020, [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/.

[27] “Deep Graph Library (DGL).” 2020, [Online]. Available: https://docs.dgl.ai/index.html#.

[28] Z. Zhang, P. Cui, and W. Zhu, “Deep Learning on Graphs: A Survey,” IEEE Trans. Knowl. Data Eng., 2020.

[29] J. Oh, K. Cho, and J. Bruna, “Advancing GraphSAGE with A Data-Driven Node Sampling,” presented at the ICLR 2019 workshop on Representation Learning on Graphs and Manifolds, 2019.

[30] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, “Graph Convolutional Neural Networks for Web-Scale Recommender Systems,” in Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, London United Kingdom, Jul. 2018, pp. 974–983, doi: 10.1145/3219819.3219890.

[31] W. Huang, T. Zhang, Y. Rong, and J. Huang, “Adaptive Sampling Towards Fast Graph Representation Learning,” in Advances in Neural Information Processing Systems, 2018, pp. 4558–4567.

[32] G. Karypis and V. Kumar, “A fast and high quality multilevel scheme for partitioning irregular graphs,” SIAM J. Sci. Comput., vol. 20, no. 1, pp. 359–392, 1998.

[33] "DGL Examples.," 2020. https://github.com/dmlc/dgl/tree/master/examples.

[34] X. Qi, "Intro Distributed Deep Learning," Intro Distributed Deep Learning, 2017. https://xiandong79.github.io/Intro-Distributed-Deep-Learning.

[35] "Distributed training with TensorFlow," 2020. https://www.tensorflow.org/guide/distributed_training.

[36] L. Ma et al., "NeuGraph: Parallel Deep Neural Network Computation on Large Graphs," in 2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19), Renton, WA, 2019, pp. 443–458, [Online]. Available: https://www.usenix.org/system/files/atc19-ma_0.pdf.

[37] Z. Tang, S. Shi, X. Chu, W. Wang, and B. Li, "Communication-Efficient Distributed Deep Learning: A Comprehensive Survey," ArXiv200306307 Cs Eess, Mar. 2020, Accessed: Nov. 20, 2020. [Online]. Available: http://arxiv.org/abs/2003.06307.

[38] S. Ghadimi, G. Lan, and H. Zhang, "Mini-batch stochastic approximation methods for nonconvex stochastic composite optimization," Math. Program., vol. 155, no. 1–2, pp. 267–305, Jan. 2016, doi: 10.1007/s10107-014-0846-1.

[39] O. Dekel, R. Gilad-Bachrach, O. Shamir, and L. Xiao, "Optimal distributed online prediction using mini-batches," J. Mach. Learn. Res., pp. 165–202, 2012.

[40] Q. Ho et al., "More Effective Distributed ML via a Stale Synchronous Parallel Parameter Server," in Advances in neural information processing systems, 2013, pp. 1223–1231.

[41] X. Jia et al., "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes," ArXiv180711205 Cs Stat, Jul. 2018, Accessed: Nov. 24, 2020. [Online]. Available: http://arxiv.org/abs/1807.11205.

[42] C.-H. Chu et al., "Efficient and Scalable Multi-Source Streaming Broadcast on GPU Clusters for Deep Learning," in 2017 46th International Conference on Parallel Processing (ICPP), Bristol, United Kingdom, Aug. 2017, pp. 161–170, doi: 10.1109/ICPP.2017.25.

[43] I. Zafar, G. Tzanidou, R. Burton, N. Patel, and L. Araujo, Hands-On Convolutional Neural Networks with TensorFlow. Packt Publishing, 2018.

[44] Jie Lu, Choon Yik Tang, P. R. Regier, and T. D. Bow, "A gossip algorithm for convex consensus optimization over networks," in Proceedings of the 2010 American Control Conference, Baltimore, MD, Jun. 2010, pp. 301–308, doi: 10.1109/ACC.2010.5530844.

[45] X. Lian, C. Zhang, H. Zhang, C.-J. Hsieh, W. Zhang, and J. Liu, “Can Decentralized Algorithms Outperform Centralized Algorithms? A Case Study for Decentralized Parallel Stochastic Gradient Descent,” in Proceedings of the 2017 Conference on Neural Information Processing Systems, 2017, vol. 30, [Online]. Available: https://proceedings.neurips.cc/paper/2017/file/f75526659f31040afeb61cb7133e4e6d-Paper.pdf.

[46] S. Boyd, A. Ghosh, B. Prabbakar, and D. Shah, “Gossip algorithms: design, analysis and applications,” in Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies., Miami, FL, USA, 2005, vol. 3, pp. 1653–1664, doi: 10.1109/INFCOM.2005.1498447.

[47] T. C. Aysal, M. E. Yildiz, A. D. Sarwate, and A. Scaglione, “Broadcast Gossip Algorithms for Consensus,” IEEE Trans. Signal Process., vol. 57, no. 7, pp. 2748–2761, Jul. 2009, doi: 10.1109/TSP.2009.2016247.

[48] F. Benezit, V. Blondel, P. Thiran, J. Tsitsiklis, and M. Vetterli, “Weighted Gossip: Distributed Averaging using non-doubly stochastic matrices,” in 2010 IEEE International Symposium on Information Theory, Austin, TX, USA, Jun. 2010, pp. 1753–1757, doi: 10.1109/ISIT.2010.5513273.

[49] P. Jesus, C. Baquero, and P. S. Almeida, “A Survey of Distributed Data Aggregation Algorithms,” IEEE Commun. Surv. Tutor., vol. 17, pp. 381–404, 2014.

[50] J. Liu, S. Mou, A. S. Morse, B. D. O. Anderson, and C. (Brad) Yu, “Request-based gossiping without deadlocks,” Automatica, vol. 93, pp. 454–461, Jul. 2018, doi: 10.1016/j.automatica.2018.03.001.

[51] A. Agarwal and J. C. Duchi, “Distributed Delayed Stochastic Optimization,” in Proceedings of the 51st IEEE Conference on Decision and Control (CDC), 2012, pp. 5451–5452.

[52] X. Lian, Y. Huang, Y. Li, and J. Liu, “Asynchronous parallel stochastic gradient for nonconvex optimization,” in Proceedings of the 28th International Conference on Neural Information Processing Systems, 2015, vol. 2, pp. 2737–2745.

[53] S. U. Stich, “Local SGD Converges Fast and Communicates Little,” presented at the International Conference on Learning Representations, 2019.

[54] F. Haddadpour, M. M. Kamani, M. Mahdavi, and V. R. Cadambe, “Local SGD with Periodic Averaging: Tighter Analysis and Adaptive Synchronization,” presented at the the 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), 2019.

[55] S. Zhang, A. Choromanska, and Y. LeCun, “Deep learning with Elastic Averaging SGD,” presented at the 3rd International Conference on Learning Representations, 2015.

[56] H. Yu, S. Yang, and S. Zhu, “Parallel Restarted SGD with Faster Convergence and Less Communication: Demystifying Why Model Averaging Works for Deep Learning,” in Proceedings of the AAAI Conference on Artificial Intelligence, 2019, vol. 33, no. 01, pp. 5693–5700.

[57] “TensorFlow.,” 2020. https://www.tensorflow.org/.

[58] “PyTorch,” 2020. https://pytorch.org/.

[59] W. Kendall, “MPI Reduce and Allreduce,” 2020. https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/.

[60] E. Yang, D.-K. Kang, and C.-H. Youn, “BOA: batch orchestration algorithm for straggler mitigation of distributed DL training in heterogeneous GPU cluster,” J. Supercomput., vol. 76, no. 1, pp. 47–67, Jan. 2020, doi: 10.1007/s11227-019-02845-2.

[61] F. Faghri, I. Tabrizian, I. Markov, D. Alistarh, D. Roy, and A. Ramezani-Kebrya, “Adaptive Gradient Quantization for Data-Parallel SGD,” presented at the the conference on Neural Information Processing Systems (NeurIPS 2020), 2020.

[62] M. Zinkevich, M. Weimer, A. Smola, and L. Li, “Parallelized stochastic gradient descent.,” in Advances in neural information processing systems 23 (2010), 2010, pp. 2595–2603.

[63] D. Kempe, A. Dobra, and J. Gehrke, “Gossip-based computation of aggregate information,” in 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings., Cambridge, MA, USA, 2003, pp. 482–491, doi: 10.1109/SFCS.2003.1238221.

[64] A. Nedic, A. Olshevsky, A. Ozdaglar, and J. N. Tsitsiklis, “On Distributed Averaging Algorithms and Quantization Effects,” IEEE Trans. Autom. Control, vol. 54, no. 11, pp. 2506–2517, Nov. 2009, doi: 10.1109/TAC.2009.2031203.

[65] “Python multiprocessing,” multiprocessing — Process-based parallelism, 2020. https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing.

[66] “DataParallel results in a different network compared to a single GPU run,” 2020. https://discuss.pytorch.org/t/dataparallel-results-in-a-different-network-compared-to-a-single-gpu-run/28635.

[67] B. Goodarzi, M. Burtscher, and D. Goswami, “Parallel graph partitioning on a cpu-gpu architecture,” in 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 58–66.