DS 256 SCALABLE SYSTEMS FOR DATA SCIENCE: PROJECT REPORT

Mitigating the Effect of Stragglers in Distributed Heterogenous Graph Neural Network Training

Pranjal Naman
PhD, Dept. of CDS
Indian Institute of Science
SR No - 21246
pranjalnaman@iisc.ac.in

***Abstract*—With the rapid increase in the size of graph data and the emergence of Graph Neural Networks (GNNs), the need for large distributed training is ever increasing. While methods exists for distributed training of GNNs, none are catered towards training on heterogeneous clusters and effectively handling stragglers. In this project, inspired by Grouped Stale Synchronous Parallel (GSSP) and Partial-Reduce (PR), we propose three novel distributed training paradigms, namely *g-ARAR*, *g-ARPS* and *mw-PR*, that mitigate the effect of stragglers in distributed training. We study the convergence, scalability and dependence on hyperparameters of the aforementioned three methods. We also make available the source code[1] for this project for further exploration.**

I. INTRODUCTION

Graph Neural Networks (GNNs) are a machine learning paradigm that involves representational learning on graph data. Graph representational learning using GNNs is used to map the high dimensional node/edge features of the graph to low dimensional embeddings (*representations*), which can be used to perform various tasks, such as node classification, graph classification, and edge prediction. These are used to perform a variety of tasks, such as node classification, graph classification, edge prediction, and more, with applications ranging from recommendation systems [1] to financial fraud detection [2].

Over the past decade, GNNs have gained significant traction from the research community. However, the attention recently has moved from building new architectures to making these architectures scalable. Despite the recent boom, the training of GNNs on large graphs remains challenging and an active area of research. The relevant data sets for GNN training have evolved into billion-scale graphs - Microsoft Academic Graph (MAG) with over 100 million nodes and over 3 billion edges and the Facebook user graph with about 2 billion nodes. With these gigantic billion-scale data sets available, it becomes imperative to focus on how to train on these graphs efficiently in a distributed setting.

Although distributed training is not a novel concept and has been employed for many years for large-scale machine learning training, it is still a very active topic of research. The process of training a model is distributed across a compute cluster and done iteratively. At the end of each epoch, the models synchronized across all processes (or a subset of processes). There are two main approaches to distributed training - *data-parallel* and *model-parallel*. The data-parallel approach involves distributing the data across the cluster, with each worker machine computing its own gradient. The model is then updated either through a centralized process, using a *Parameter Server*, or through a decentralized process, using a ring *all-reduce*. The model-parallel approach, on the other hand, involves splitting the layers of the model across the cluster and distributing the computations.

For large-scale GNN training, a *data-parallel* approach is usually employed [3], [4]. The graph is partitioned into subgraphs and distributed across workers. Each worker then trains their own copy of the model independently, synchronizing with other workers intermittently using a *decentralized* approach using `ring all-reduce`. The synchronization time at the end of each epoch is bottle-necked by the slowest processes, which are referred to as *stragglers*. The effect of the stragglers on the training time is exacerbated when training is done on heterogeneous systems, where workers cannot move on to the next epoch will the slowest process has reached the collective communication call.

As stated my Miao et al in [5], heterogeneity in real-world systems is primarily caused by the following:

- **Data & Communication Heterogeneity** - Data may not be distributed equally amongst workers and resulting in each worker having different per epoch time. Moreover, poor network connectivity between worker nodes can lead to delays in communication.
- **Hardware Heterogeneity** - GNN training is compute-intensive and relies on hardware accelerators. These accelerators are developing rapidly, sometimes resulting in different hardware on different nodes of the same cluster. Since different generations of accelerators have distinct computational powers, collaborative training with these accelerators could lead to hardware heterogeneity.
- **Resource sharing** - A common practice nowadays is to utilize cluster schedulers, to manage clusters of CPU/GPU resources. The jobs could, therefore, differ in terms of resource and memory utilization, network bandwidth etc., thus introducing heterogeneity in com-

[1] https://github.com/pranjalnam/SSDS-Project-2023

pute times.

While the existing frameworks for GNN training like DistDGL [3] and PyTorch Geometric [4] provide distributed *data-parallel* training techniques, none to the best of our knowledge focus on mitigating the effect of stragglers. In this project, we study the effects of stragglers on synchronous as well as asynchronous training methods. Further, inspired by the existing work on grouping workers, we group workers of similar epoch times and implement three techniques that try to help mitigate the effect of stragglers while retaining the benefits of both completely synchronous and completely asynchronous training methods. These methods are synchronous on the group level and asynchronous (either partially or completely) on a global basis.

This report is structured as follows: Section(II) presents the current state of distributed training research and introduces two methods that this project is inspired by - namely *Grouped Stale Synchronous Parallel (GSSP)* method [6] and *P-Reduce (PR)* method. Section(III) introduces the three methodologies this project proposes - *Grouped All-Reduce All-Reduce (g-ARAR)*, *Grouped All-Reduce Parameter Server (g-ARPS)* and *Master-Worker P-reduce (mw-PR)*, and goes into the details of the implementation of each method. Section(IV) details the experimental setup in subsection(IV-A) and describes the experiments conducted in order to study the behavior of each method in detail in subsections(IV-B,IV-C, IV-D). Finally, in section(V), we summarize the our methodologies, the experiments we have conducted and have a brief discussion of the future works.

II. Related Work

With the rise in the volume of data in the past few decades and increasingly complex models in order to learn the data, training neural networks on a single machine has become essentially impossible. This has made computation and communication efficient distributed and accelerated training strategies essential. Most distributed training strategies follow a *data-parallel* paradigm, where the same model is trained independently on different workers, each of which has a subset of the data. For the accumulation of the gradients from each worker, two primary strategies are used: asynchronous *centralized* strategies using *parameter servers* (PS) [7], and synchronous *de-centralized* strategies using `ring all-reduce` [8].

Dean et al. in their seminal 2012 paper [9], proposed a method that allows for the training of a deep neural network with billions of parameters on tens of thousands of CPU cores. This was done using a centralized asynchronous parallel SGD algorithm that aggregates the model parameters to a group of *parameter servers*. Multiple workers process mini-batches of the data that they own independently and simultaneously and asynchronously update the parameter servers and continue training using the values that the parameter server previously held.

The PS paradigm has two major drawbacks. The first is that for a large number of workers, each of which will try to update the parameters, the PS becomes a communication bottleneck. The second problem is of the staleness of the values in the PS. This problem becomes particularly egregious in heterogeneous environments, where there is a significant difference between the per epoch times of the fastest and slowest worker. This means that sometimes the fastest workers, which would have trained much more epochs than the slowest workers, will be trained with stale parameter values from the slow workers whenever the slow workers update the PS. This leads to slow model convergence. Further, it has been shown that for asynchronous-SGD, there are complex inter-dependencies between model hyper-parameters and implementation details like the number of workers, etc., impacting the convergence time of the algorithm [10].

On the other side of the spectrum, we have a synchronous decentralized method that requires all the workers to average the newly computed gradients with the help of a *ring all-reduce*. This paradigm requires all the workers to move in lock-step with each other, causing all workers to wait for the slowest worker every epoch. This leads to rapid convergence in training. However, this method also has two major drawbacks. Firstly, every iteration is bottle-necked by the slowest worker, since all workers will wait in the all-to-all collective *all-reduce* call. This is a major drawback for training on heterogeneous clusters. Secondly, this method is completely intolerant to devise drop-outs, which are a common occurrence in large commodity clusters.

Multiple attempts have been made to establish a paradigm that keeps the advantages of both the asynchronous centralized method and synchronous de-centralized method while mitigating the disadvantages of both. As a middle-ground for synchronous and asynchronous algorithms, *Stale Synchronous Parallel* algorithms were also introduced, in which the iteration difference between the fastest and the slowest worker is limited by a pre-assigned threshold. Workers wait in the *all-reduce* only till this threshold time is reached and then execute the *all-reduce* operation. In practice, however, it is only suitable for homogeneous clusters.

Multiple specialized algorithms have been proposed to efficiently train deep networks on heterogeneous clusters. Sun et al. [6] proposed a centralized asynchronous *Grouping Stale Synchronous Parallel* algorithm, where workers of similar performance are grouped together with the use of *Jenks Natural Breaks* [11] algorithm and each group is assigned a *group server*. Each group server is responsible for accumulating gradients from workers in its group, periodically synchronizing with other group servers, which is done with the help is a master parameter server. This reduces the effect of stale gradient due to stragglers since all workers within a group will have similar performance, thus making this scheme ideal for heterogeneous clusters.

Miao et al. [5] proposed a decentralized asynchronous method based on `all-reduce`. Here, only p out of all the workers have to synchronize at a time. Thus, when a worker finishes computing the gradients, it waits for (p-1) other workers to have computed their gradients. Once a set of p workers are ready with their gradients, they aggregate

their gradients with the use of a `ring all-reduce` operation. As training proceeds, gradient updates from each worker slowly propagate to all other workers and allow all the workers to collaboratively reach convergence. By introducing asynchrony into the `all-reduce` paradigm, this scheme reduces its sensitivity to stragglers. The authors have shown that the PR scheme performs well on heterogeneous clusters.

Distributed training of GNNs is a rapidly emerging field of representational learning on graphs. *Pytorch Geometric* [4] and *DistDGL* [3] are the two leading frameworks that support distributed GNN training on graphs. Parameter synchronization on both these frameworks happens in a decentralized fashion. *DistDGL* is the state-of-the-art framework for distributed GNN training. It uses synchronous SGD for dense parameters and an asynchronous SGD in the Hogwild fashion for the sparse vertex embeddings. *Pytorch Geometric*, the GNN framework, built on top of Pytorch [12], provides distributed training support that used *DistributedDataParallel*, with synchronous SGD for parameter synchronization.

III. Methodology

The aim of this project is to propose and evaluate methods that can efficiently train GNNs on massive datasets in a distributed setting using a heterogeneous cluster of systems. The three methods that we propose are inspired by *GSSP* [6] and *P-Reduce* [5], the details of which are given in section(II). All three methods aim to achieve the following ideas:

1) The proposed methods try to keep the benefits from both the synchronous and the asynchronous methods while trying to mitigate their disadvantages.
2) The methods are optimized to run in a distributed manner on heterogeneous devices, where there is a wide difference in the per-epoch time between the fastest and the slowest workers.

The second point is easily addressed by grouping workers together in terms of their per-epoch computation time, inspired by GSSP, and synchronizing the groups in a centralized or decentralized manner. However, unlike GSSP, we propose to do the inter-group updates in a decentralized manner. This is an obvious choice, since in a group, the per-epoch time difference between the fastest and the slowest worker will be small. Thus it makes sense to take advantage of the convergence properties of a decentralized method by applying it intra-group. The inter-group updates may, again, be done in a centralized (*g-ARPS*) or decentralized (*g-ARAR*) fashion. In this project, we look at both variations of the method.

The same issues may be addressed using the *P-reduce* method [5]. However, the *P-reduce* implementation presented in [5], is done on the GPU and utilizes the GPU shared memory. We implement a distributed version of the *P-reduce* method using a Master-Worker strategy (*mw-PR*).

With the data hash partitioned equally across all workers, we use `MPI` directive exposed by `torch.distributed` to implement the three aforementioned methods, the details of which are given below. Before the training iterations start, workers are divided into a predetermined number of groups with the use of a lightweight profiler script. We now discuss the proposed methodologies in detail.

`Grouped All-reduce All-reduce(g-ARAR)`:

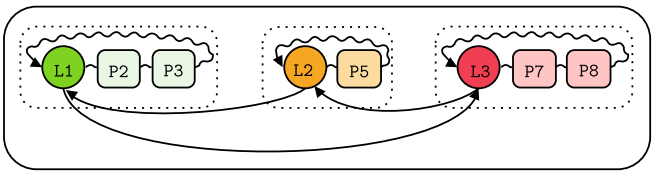


Fig. 1: Schematic of `g-ARAR` method

In this method, training is done in a completely decentralized manner (Fig 1). Each worker trains their copy of the model independently with the subset of data (subgraph, in our case) they have been assigned. At the end of every epoch, all workers in a group average their gradients with the help of a `dist.all_reduce()`. In order to aggregate all the model gradients with one all-reduce call, all the gradient tensors are flattened and concatenated into one tensor, on which the all-reduce is called. Once each worker in a group has the aggregated gradients, they calculate the average gradient, reshape the aggregated tensor into the correct shape, update the model gradients with the averaged gradients, and move on to the next epoch. Every pre-determined interval of time (45 seconds in our case), the group leaders (which in our case is the lowest ranked process in each group; `L1`, `L2` & `L3` in Fig 1) enter an all-reduce call and calculate their average model parameters and update their respective models. In order to do this in one all-reduce call, all leader workers, again, flatten out the model parameter tensors. Once the model parameters of the leader processes are updated, they move on to the next epoch. The updates that the group leaders receive from the other group leaders slowly pass onto the worker processes in each group due to the inter-group all-reduce.

This method mitigates some of the waiting due to slow processes by eliminating all-to-all all-reduce calls amongst the workers, thus reducing the wait time of the fast processes significantly. Further, this method eliminates staleness in the parameter updates. The downside of this method is that it is not tolerant to worker dropout.

`Grouped All-reduce PS(g-ARPS)`:

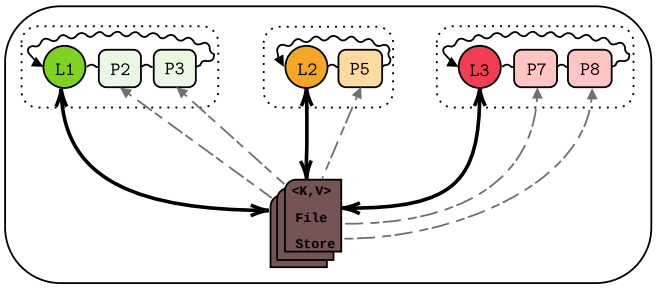


Fig. 2: Schematic of `g-ARPS` method

In this method, training is done in a decentralized manner intra-group and a centralized manner inter-group. Each worker trains their copy of the model independently with the subset of data they have been assigned. At the end of every epoch, all workers in a group average their gradients with the help of a `dist.all_reduce()`. Like *g-ARAR*, in order to aggregate all the model gradients with one all-reduce call, all the gradient tensors are flattened and concatenated into one tensor, on which the all-reduce is called. Once each worker in a group has the aggregated gradients, they calculate the average gradient, reshape the aggregated tensor into the correct shape, update the model gradients with the averaged gradients, and move on to the next epoch. For the intra-group synchronization, every group is assigned a parameter server. Every pre-determined interval of time (45 seconds in our case), the group leaders (which in our case is the lowest ranked process in each group, `L1`, `L2` & `L3` in Fig 2) put their model parameters in their group's parameter server. Every process then calculates the average of the parameters stored in each group's parameter servers, updates their own model, and moves on to the next epoch. The parameter server in our case is implemented using as a distributed key-value store (`dist.FileStore`), with the group name as the key and the model parameters as values. When it's time to update the model parameters, each worker simply fetches the current parameter values in the `FileStore` corresponding to each group.

The primary advantage of this method is that it is asynchronous on an inter-group basis and will perform well for systems where workers have variable compute times. However, this still has the problem of stale updates, although mitigated somewhat by maintaining a different parameter server for each group and taking an average of the parameter values for all groups every time interval.

`Master-Worker Partial-Reduce(mw-PR)`

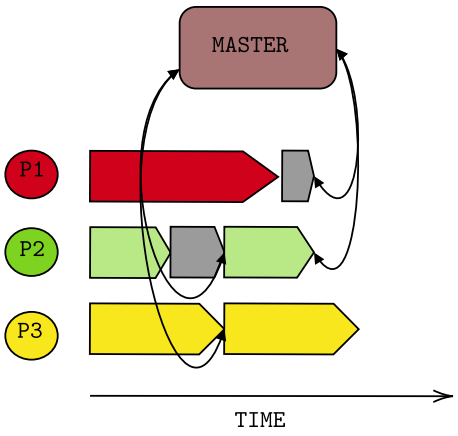


Fig. 3: Schematic of `mw-PR` method

In the partial all-reduce proposed by Miao et al. [5], all workers train their model independently with their subset of the data. After every epoch, the workers enter an all-reduce call like the traditional decentralized method. In the *P-reduce* method, as soon as P processes call the all-reduce, the operation is executed and the processes move on to their next epoch. This is not trivially done in a distributed setting with `MPI` point-to-point and collective directives. We implement a distributed version of this method with the use of a *master* process. The master process posts a set of P `dist.recv()` calls at the beginning of every iteration of an infinite `while` loop and appends the ranks of the processes from which it receives the parameters to a list. The sending worker process appends its rank to the flattened model parameter tensor. Once it has received parameters from P processes, the master process averages the received parameters and sends the averaged parameter value to the processes that have sent their updates and whose ranks were saved. Each worker process, at the end of every epoch, sends the new model parameters to the master process, with its rank piggy-backed onto the flattened tensor. It then waits for the master to send it the updated averaged parameters, which it uses to update its model parameters.

	Nodes ($\|V\|$)	**Edges ($\|E\|$)**	**I/p features (d)**	**Num Classes ($\mathcal{C}$)**
cora-full	19.8K	126.8K	8710	70
reddit	232.9K	114.6M	602	41

TABLE I: Data sets used for evaluation

This method avoids any kind of grouping of workers and still achieves our primary goals, namely minimizing staleness and minimizing waiting times of workers. This method is also tolerant to the dropout of workers, and also gives the best performance of the three in terms of convergence.

We implement the three methods above described methods, namely *g-ARAR*, *g-ARPS* and *mw-PR* using `MPI` point-to-point communication directives (`dist.send`, `dist.recv`), collective communication directives (`dist.all_reduce`, `dist.broadcast`) and using distributed key-value store (`dist.FileStore`) exposed by `torch.distributed` API. These methods are used in conjunction with `DGL` to train a vanilla GCN model on two data sets of varying sizes, mentioned in table(I).

IV. Experiments and Results

In this section, we evaluate the three proposed methods, namely *g-ARAR*, *g-ARPS* and *mw-PR*, and present the results for the same. For each method, we compare how fast the model converges, the parameters model is dependent on (p in *mw-PR*), and the method's scalability. Subsection(IV-B) presents the model convergence trends for each method. Subsection(IV-D) presents the scalability studies. Subsection(IV-C) explores how model convergence trends change with p for *mw-PR*. Each subsection goes into a brief analysis of the results it presents. But first, we present the experimental setup on which all the experiments were run on.

A. *Experiment Setup*

All experiments were conducted on the **IoE Cluster** which has the following node specifications:

- **Node 0** (master) - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM.
- **Node 1** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM.
- **Node 2** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM.
- **Node 3** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM.
- **Node 4** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM, $2\times$ Titan RTX GPUs with 24GB of memory each.
- **Node 5** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM, $2\times$ Titan RTX GPUs with 24GB of memory each.
- **Node 6** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM, $2\times$ Titan RTX GPUs with 24GB of memory each.
- **Node 7** - $2\times$ AMD EPYC CPUs with 16 physical cores each, 256GB of RAM, $2\times$ Titan RTX GPUs with 24GB of memory each.
- **Node 9** - Intel Xeon CPU with 32 physical cores, 192GB of RAM, $4\times$ Tesla V100 GPUs with 32GB of memory each.

For all the experiments excluding the scalability study, the number of workers is set to 8 with 1 worker running on each compute node. The heterogeneity is introduced using two methods - introducing random sleeps during training and changing worker processes' niceness (priority) using the `os.nice()` API.

We employ a vanilla `GraphConv` model built using the `DGL` library to evaluate our proposed methods. The model comprises two GraphConv layers. The data sets used for the experiments are hash partitioned onto the worker nodes using the simple hash function - $key\%\#workers$. The partitioned subgraphs are comprised of two types of nodes - the nodes local to the subgraph and *halo* nodes introduced due to edge cuts.

B. *Model Convergence*

In a homogeneous training environment, where stragglers are absent, it has been shown in literature that the synchronous decentralized training method using *all-reduce* outperforms all other training strategies in terms of convergence. This is expected, as in synchronous scenario, every worker gets updates from every other worker, leading to rapid converge of the model. However, if stragglers are present in the system, the synchronous method suffers greatly. Since all workers have to wait for the slowest worker in every epoch, the per-epoch time is greatly increased, leading to slow convergence of the model. Figures(4) show the above described deterioration of the convergence rate of the synchronous decentralized method due to the effect of stragglers. For the *Cora* dataset, *all-reduce* with stragglers (s-AR) takes about twice as long as *all-reduce* without stragglers (s-AR), to reach the 62.5% accuracy. For the *Reddit* dataset, the time is even more. Thus, for the purpose of bench-marking the proposed methods, we consider the s-AR as the **baseline method** and AR as the **roofline method**.

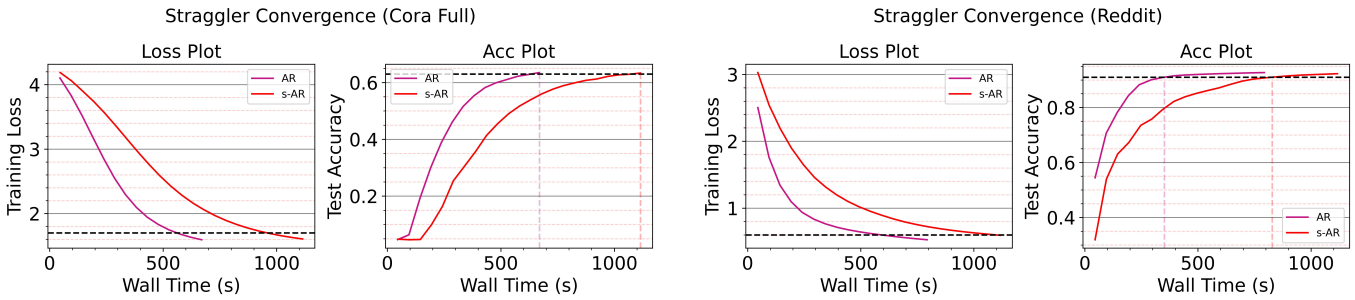
Figures(5) present the evolution in the Training Loss and the Model Accuracies for for all proposed methods, namely *g-ARAR*, *g-ARPS* and *mw-PR* (with $p = 2$), along with *AR* (roofline) and *s-AR* (baseline) mdoels. For *mw-PR*, $p = 2$ is chosen since the method performs the best for this value of p. As expected, the performance of *g-ARAR*, *g-ARPS* and *mw-PR* methods lie somewhere between the *roofline* model performance and the *baseline* model performance. For both the datasets, *mw-PR* performs the best out of the three proposed methods, followed closely by *g-ARAR*. Both these methods only marginally lag behind the roofline *AR* method. While $g - ARPS$ lags behind the other proposed methods, it is still significantly faster than the baseline method.

Since in the *g-ARPS* method, inter-group updates are done using a parameter server, the method still suffers from stale parameter updates, causing slower convergence. This is despite the fact that in the *g-ARPS* method, processes spend the least amount of time waiting and thus allowing the model to train the most number of epoch out of the three proposed methods. In the *g-ARAR* methods, the group leaders synchronized based on elapsed wall-clock time. This reduces the waiting time of the workers, since clocks of all workers are independent and are expected to show approximately the same elapsed time. Although minimal, *g-ARAR* still incurs waiting times for the faster workers. Even so, the method converges almost as quickly as the roofline method. This is because the synchronous updates on the inter-group level eliminates all staleness. Thus, the slight decrease in the convergence rate *g-ARAR* may be attributed to the waiting time incurred in the synchronous updates.

However, both he staleness issue as well as the wait time is addressed by the *mw-PR* method. For small values of p and a large number of workers, there waiting time for workers wanting to synchronize is expected to be minimal and since all workers eventually participate in the update aggregation, the problem of staleness is mostly mitigated. These properties of the algorithm allow it to have convergence rates comparable to the roofline method. However, this method is very sensitive to the value of p, whose optimum value must be found before this method is employed.

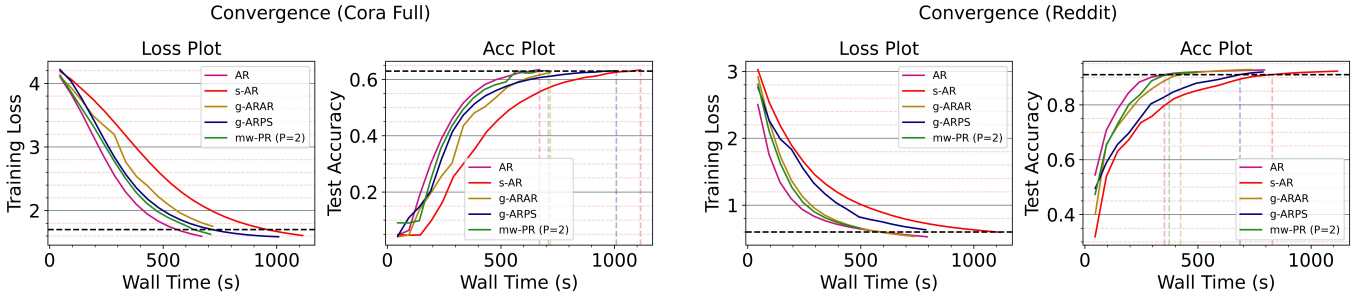
C. *Effect of p in mw-PR*

In *mw-PR*, p workers send their updated parameters to the master process after every batch. Intuitively, the *mw-PR* is expected to sensitive to the value of p. As $p \rightarrow \#workers$ present, the *mw-PR* approaches the baseline method. For $p = 1$, the convergence rates are also expected to be poor since *mw-PR* will then act as an un-grouped traditional parameter server, which also has poor performance due to staleness. Thus, one has to find the value of p which has the best trade-off between



(a) (b)

Fig. 4: (a)Training Loss and Test Accuracies obtained while training the described GCN model on the Cora Full Dataset for *all-reduce* in the absence of stragglers (roofline) and all-reduce in the presence of stragglers (baseline) methods with 8 workers. (b)Training Loss and Test Accuracies obtained while training the described GCN model on the Reddit Dataset for *all-reduce* in the absence of stragglers (roofline) and all-reduce in the presence of stragglers (baseline) methods with 8 workers.



(a) (b)

Fig. 5: (a)Training Loss and Test Accuracies obtained while training the described GCN model on the Cora Full Dataset for all proposed methods along with the baseline and the roof-line methods, trained using 8 workers. (b)Training Loss and Test Accuracies obtained while training the described GCN model on the Reddit Dataset for all proposed methods along with the baseline and the roof-line methods, trained using 8 workers.

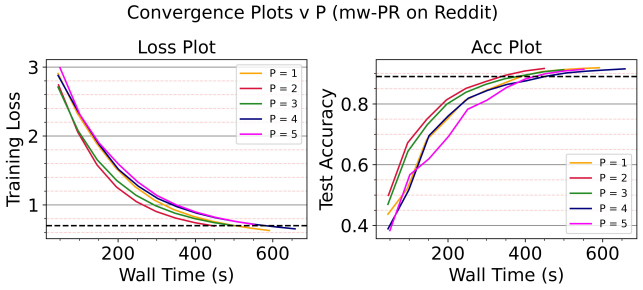


Fig. 6: Plot showing the convergence of *mw-PR* for various values of p.

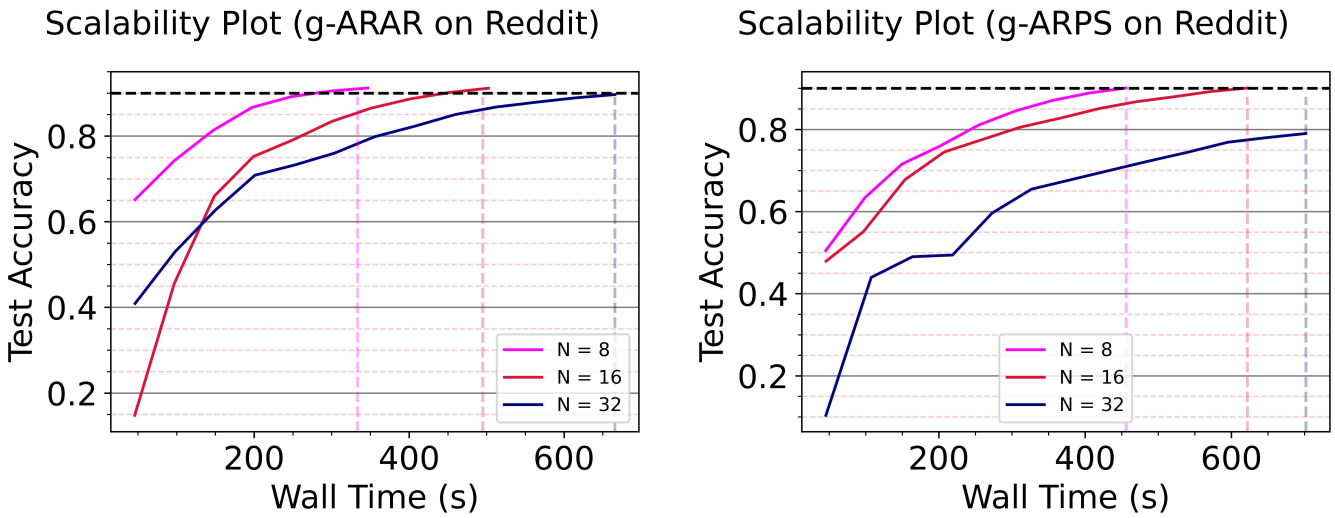
wait-time and receiving sufficient updates in order to get the best performance from the method.

Figure(6 reports the evolution of Training Loss and Test Accuracy for $p = \{1, 2, 3, 4, 5\}$ for 8 workers. For the Reddit dataset using the aforementioned GCN model, we see that $p = 2$ has the best convergence rate. The convergence rate slowly decreases for $p > 2$, since the waiting time of the workers increase with the increase in p. It is obvious that the $mw-PR$ model is task dependent and is expected to perform best for some other value of p for other hyperparameters of the problem (like a different model or a different number of workers). Even so, the optimal value of p is expected to be small ($p << \#workers$), since waiting time is the primary cause of performance degradation in this case.

D. *Scalability Study*

One of the primary metrics that all distributed paradigms are judged on is scalability. For a method to be practically usable in a data parallel setting, it is expected to strongly scale with increase in the number of processes. We study the strong scaling performance of *g-ARAR*, *g-ARPS* and *mw-PR* with increase in the number of available workers.

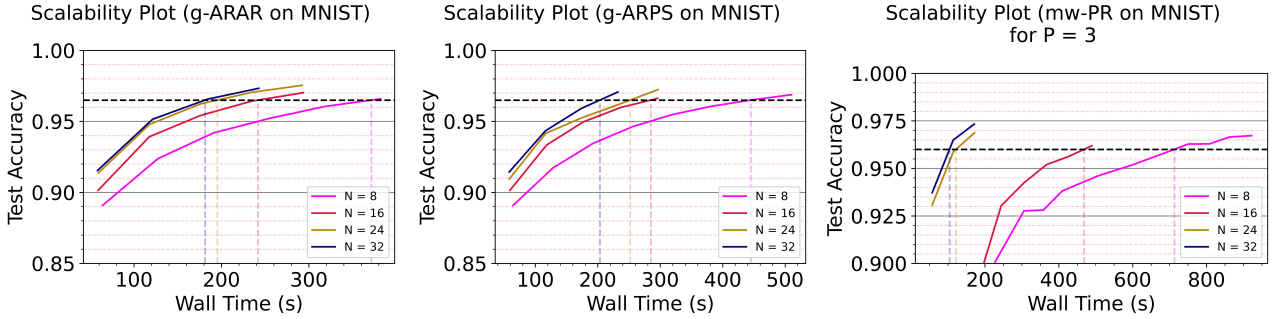
The results of the scalability study on Reddit dataset is reported in Figure(7). For the aforementioned GCN model trained on the Reddit dataset, we observer that methods *g-ARAR* and *g-ARPS* scale very poorly and exhibit negative scaling curve. Upon further investigation, we come to the conclusion that due to insufficient computational work (due to small model and data size), it is the communication overhead that dominates over the computation as the number of workers is increased, causing negative scaling. We further verify our



(a) Scalability study of *g-ARAR* for the GCN model described in subsection IV-A trained with the Reddit dataset.

(b) Scalability study of *g-ARPS* for the GCN model described in subsection IV-A trained with the Reddit dataset.

Fig. 7



(a) (b) (c)

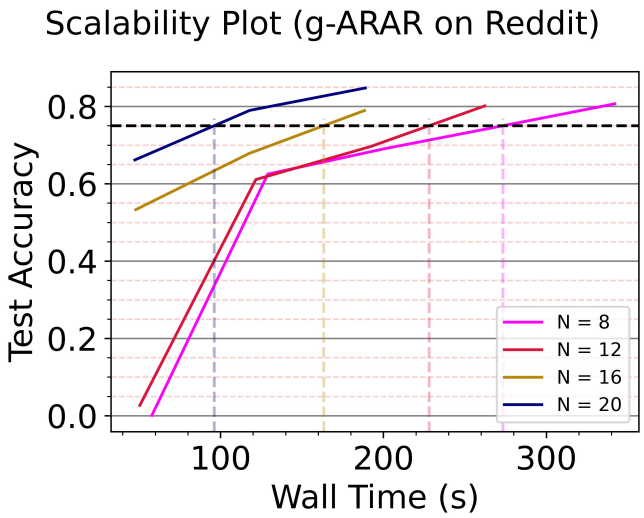
Fig. 8: Scalability study of *g-ARAR*, *g-ARPS* and *mw-PR* for a DNN model trained on MNIST data.

aforementioned hypothesis by implementing the two methods on large DNN training workload and observe that all three proposed methods exhibit good strong scaling up-to a certain number of workers, shown in Figure(8). To further underscore this observation, we increase the complexity of our GNN model (to increase the computational work) and rerun the scalability study on the Reddit dataset. The results are reported in Figure(9). in this case, we observe that it is computation that dominates over the communication and resulting in strong scaling.

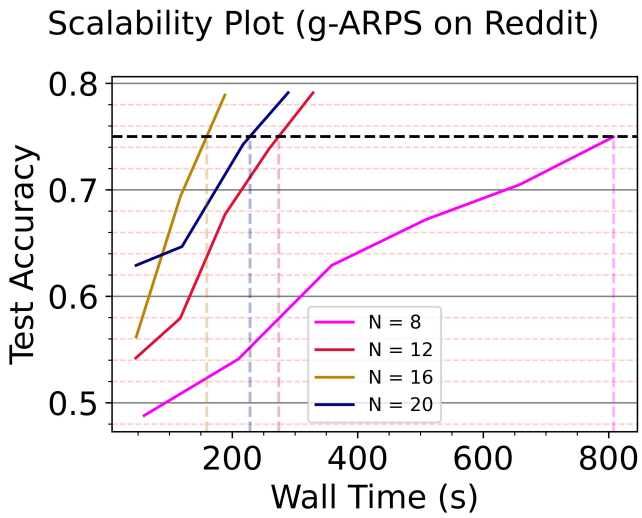
V. Conclusions & Future Work

Distributed training is a generic paradigm that can be applied to variety of applications, ranging from vanilla distributed SGD to a federated learning setup. In this project, we study the role that stragglers and staleness play in distributed GNN training framework. We propose three methodologies, namely, *g-ARAR*, *g-ARPS* and *mw-PR* that address both the aforementioned issues to different magnitudes. All three methods hold merit and could find use in distributed training depending on the end goal. *g-ARAR* uses global synchronization but does not suffer from the staleness problem due to its locally and globally synchronous nature. With the aim of increasing work efficiency of the involved workers to the maximum, one might pick *g-ARPS* since the waiting time is negligible owing to no inter-group synchronizations. On the other hand, *mw-PR* deals with the staleness issue better than the other two methods and can lead to a faster convergence.

With the above insights in mind, the methodologies proposed in this project can find various applications. We intend to expand this work into performing GNN training on standalone subgraphs where there is no communication involved across workers for sampling neighbours and each subgraph is trained independently on. In addition to this, we also plan to take this work and apply it to more complex distributed training paradigms such as federated GNNs. Since the advent of distributed training for machine learning workloads, there has been a constant tussle between reducing staleness and



(a) Scalability study of *g-ARAR* for the augmented GCN model described in subsection IV-A trained with the Reddit dataset.



(b) Scalability study of *g-ARAR* for the augmented GCN model described in subsection IV-A trained with the Reddit dataset.

Fig. 9

mitigating the effect of stragglers. This work is a step towards achieving a balance distributed training framework.

REFERENCES

[1] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–37, 2022.

[2] Y. Liu, X. Ao, Z. Qin, J. Chi, J. Feng, H. Yang, and Q. He, "Pick and choose: A gnn-based imbalanced learning approach for fraud detection," in *Proceedings of the Web Conference 2021*, ser. WWW '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3168–3177. [Online]. Available: https://doi.org/10.1145/3442381.3449989

[3] D. Zheng, C. Ma, M. Wang, J. Zhou, Q. Su, X. Song, Q. Gan, Z. Zhang, and G. Karypis, "Distdgl: Distributed graph neural network training for billion-scale graphs," *CoRR*, vol. abs/2010.05337, 2020. [Online]. Available: https://arxiv.org/abs/2010.05337

[4] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR 2019 Workshop on Representation Learning on Graphs and Manifolds*, 2019. [Online]. Available: https://arxiv.org/abs/1903.02428

[5] X. Miao, X. Nie, Y. Shao, Z. Yang, J. Jiang, L. Ma, and B. Cui, "Heterogeneity-aware distributed machine learning training via partial reduce," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21. Association for Computing Machinery, 2021, p. 2262–2270.

[6] H. Sun, Z. Gui, S. Guo, Q. Qi, J. Wang, and J. Liao, "Gssp: Eliminating stragglers through grouping synchronous for distributed deep learning in heterogeneous cluster," *IEEE Transactions on Cloud Computing*, vol. 10, no. 4, pp. 2637–2648, 2022.

[7] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. USA: USENIX Association, 2014, p. 583–598.

[8] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

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

[10] W. Zhang, S. Gupta, X. Lian, and J. Liu, "Staleness-aware async-sgd for distributed deep learning," in *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, ser. IJCAI'16. AAAI Press, 2016, p. 2350–2356.

[11] G. F. Jenks, "The data model concept in statistical mapping," *International yearbook of cartography*, vol. 7, pp. 186–190, 1967.

[12] S. Li, Y. Zhao, R. Varma, O. Salpekar, P. Noordhuis, T. Li, A. Paszke, J. Smith, B. Vaughan, P. Damania, and S. Chintala, "Pytorch distributed: Experiences on accelerating data parallel training," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 3005–3018, aug 2020. [Online]. Available: https://doi.org/10.14778/3415478.3415530