Distributed Graph Neural Network

University of Pisa - Computer Science
Jacopo Raffi
Prof. Patrizio Dazzi

01/08/2024 - 30/10/2024

Contents

1 Introduction

This study investigates distributed training methods to improve the computational performance of training Graph Neural Networks (GNNs) in a distributed setting, using a GNN applied to image classification tasks as a study case. In particular, the investigation focuses on the effectiveness of pipeline parallelism, as well as the combination of data parallelism with pipeline parallelism, in improving the efficiency of the training process. The accuracy of the model is not the focus of this work. Additionally, this study aims to leverage existing libraries to implement these parallelism strategies, considering that PyTorch Geometric (PyG) does not currently support model parallelism for graph neural networks.

Section 2 discusses the sequential model and evaluates its performance in the context of image classification. Section 3 focuses on pipeline parallelism, evaluating a two-stage and a four-stage approach to improve training efficiency. Section 4 examines the integration of data parallelism with pipeline parallelism and evaluates its impact on overall model performance. Section 5 provides an overview of the technical setup, including the hardware and software configurations used in the study. Finally, Section 6 provides a summary of the entire work and discusses possible future directions, including both distributed training and inference of graph neural networks.

2 Sequential

The image classification model, evaluated on the CIFAR-10 dataset, used in this study is based on the approach described in [2] with slight differences; the input image is not divided into patches because CIFAR-10 images are relatively “simple" compared to more complex datasets such as ImageNet.

The ViG model consists of ViG blocks. An individual ViG block consists of two main components:

- The Grapher module applies a linear layer before and after the graph convolution to project the node features into the same domain and increase the feature diversity. This can be formulated as $Y = \sigma(GConv(XW_{in}))W_{out} + X$;
- The FFN module is a simple multi-layer perceptron with two fully-connected layers. It can be expressed as $Z = \sigma(YW_1)W_2 + Y$.

The evaluation of training performance was conducted using 100 minibatches, each with a size of 1000, drawn from the 50,000-image training dataset. For testing, performance was measured on 20 minibatches from the 10,000-image test dataset, with the primary focus being on training performance. The test performance is included solely for illustrative purposes. The tables 1, 2 and 3 present the various statistics computed for the whole model, for a single ViG block and for the readout, respectively.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	37.44	0.31	36.98	38.85
Test	13.56	0.042	13.47	13.62

Table 1: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	1.95	0.038	1.91	2.21
Test	1.68	0.010	1.67	1.74

Table 2: Summary statistics (seconds) for a single ViG block.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	0.011	0.0005	0.010	0.013
Test	0.0105	0.0001	0.010	0.0109

Table 3: Summary statistics (seconds) for the readout.

The results show that, as expected, the computation time is mainly determined by the execution of the ViG blocks. The readout phase contributes negligibly to the total computation time. This is expected since the readout consists of a simple multi-layer perceptron (MLP) with only two layers and the output consists of only 10 units.

3 Pipeline Parallelism

Pipeline parallelism is a technique that partitions a model's computation into sequential stages, allowing these stages to execute concurrently across multiple devices. This approach is particularly effective for improving training efficiency.

GPipe[3] introduces an innovative approach to pipeline parallelism, specifically designed to facilitate the efficient training of large-scale neural networks. This method addresses the challenges posed by growing model size and complexity by dividing the neural network into multiple stages, referred to as "cells," and allocating these stages across numerous computational nodes. GPipe employs a technique known as micro-batching, in which each mini-batch of training data is further subdivided into smaller micro-batches. These micro-batches are processed independently through the pipeline, facilitating synchronous mini-batch gradient descent during the training process. Gradients are accumulated across all micro-batches within a mini-batch and applied collectively at the end of the mini-batch. This approach ensures that gradient updates remain consistent regardless of the number of micro-batches.

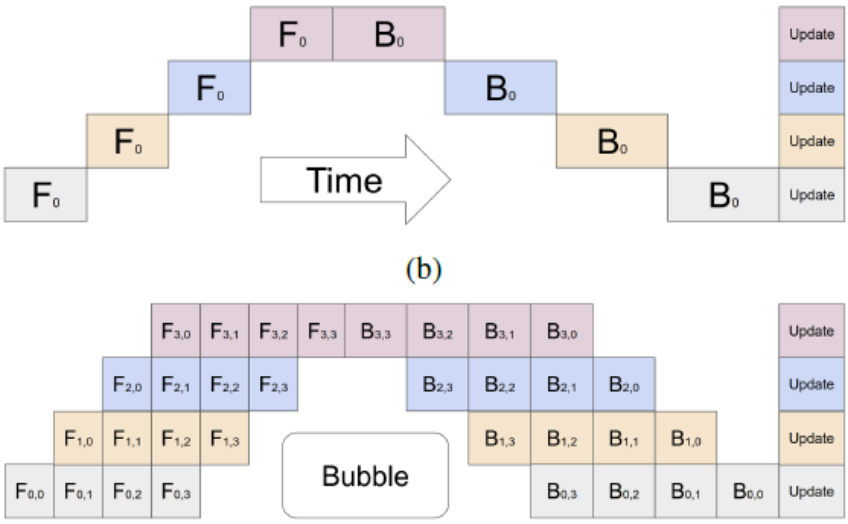

Figure 1: Pipeline parallelism divides the input mini-batch into smaller micro-batches, enabling different nodes to work on different micro-batches simultaneously. Gradients are applied synchronously at the end.

To explore this approach with the sequential model used in this study, pipeline parallelism was implemented with 2 and 4 stages and with 5 and 10 microbatches for each configuration. For the 2-stage configuration, 4 blocks

were assigned to each stage, with the readout part in the second stage, which has a negligible computational time. For the 4-stage configuration, the model was divided into 2 blocks per stage. In this setup, the readout part still resides in the last stage, maintaining its minimal computational cost. Similar to the sequential model, the focus here was primarily on the training phase, with performance statistics based on 100 training minibatches, while only 20 test minibatches were used for illustrative purposes.

Tables 4 and 5 show the results obtained using the 2-stage pipeline model.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	25.69	1.03	19.59	27.06
Test	8.19	1.76	7.53	13.87

Table 4: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches, each minibatch split in 10 micro-batches, using a 2-stage pipeline.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	19.02	0.10	18.94	19.92
Test	7.25	0.02	7.21	7.28

Table 5: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches, each minibatch split in 5 micro-batches, , using a 2-stage pipeline.

With 2 pipeline stages, the speedup achieved with 10 microbatches was about 1.45, while with 5 micro-batches the speedup increased to about 1.96. These results suggest that using fewer microbatches leads to better performance in this pipeline configuration, possibly due to a reduction in communication overhead. It is important to note that with 10 microbatches, there are 10 microbatches for the forward pass and 10 for the backward pass, for a total of 20 microbatches. In contrast, with 5 microbatches, there are only 10 microbatches in total, which could result in similar overhead during the “forward" phase of the 10 microbatch configuration.

Building on the results from the 2-stage pipeline, the performance of the 4-stage pipeline was also evaluated. Tables 6 and 7 show the results obtained using the 4-stage pipeline model.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	29.36	0.17	28.91	29.63
Test	11.08	0.04	11.02	11.16

Table 6: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches, each minibatch split in 10 micro-batches, using a 4-stage pipeline.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	28.12	0.09	27.99	28.57
Test	10.63	0.03	10.57	10.69

Table 7: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches, each minibatch split in 5 micro-batches, using a 4-stage pipeline.

The results of the 4-stage pipeline showed significantly worse performance compared to the 2-stage approach, with a speedup of 1.29 for 10 microbatches and 1.33 for 5 microbatches. However, these results are still slightly better than the sequential model. While this approach may not provide the best performance under the current fixed model parameters, it could still be useful in different scenarios. Adjusting the model parameters could potentially lead to improved performance. In addition, even with the lower speedup, the 4-stage pipeline may still provide advantages in terms of memory usage, as breaking the model into smaller stages allows for more efficient memory usage.

For pipeline parallelism, there is no need to check whether the loss is different from the sequential model due to the GPipe implementation. The use of microbatches in GPipe does not affect the calculation of gradients, ensuring that the gradients remain consistent with those of the sequential model. Consequently, the final loss is unaffected, maintaining equivalence with the results of the sequential model.

4 Combined Data and Pipeline Parallelism

The “Data + Pipeline parallelism" approach in this study involves using two identical copies of the model, with each copy divided into stages according to the pipeline parallelism method discussed earlier. The data set is split in half, with each copy receiving a portion of the data. The 2-stage configuration was chosen

because of its proven efficiency. To ensure proper synchronization between the stages of each model copy, the *DistributedDataParallel* class of PyTorch was used, with synchronization occurring at each minibatch. A graphical representation of this approach is shown in Figure 2. The goal of the study is to evaluate the impact of the overhead introduced by data parallelism on the computation of minibatches, with a focus on the training phase. The objective is to determine whether combining data parallelism with pipeline parallelism can further improve the computational performance of the model. Tables 9 and 8 show the results obtained with 5 and 10 microbatches, respectively:

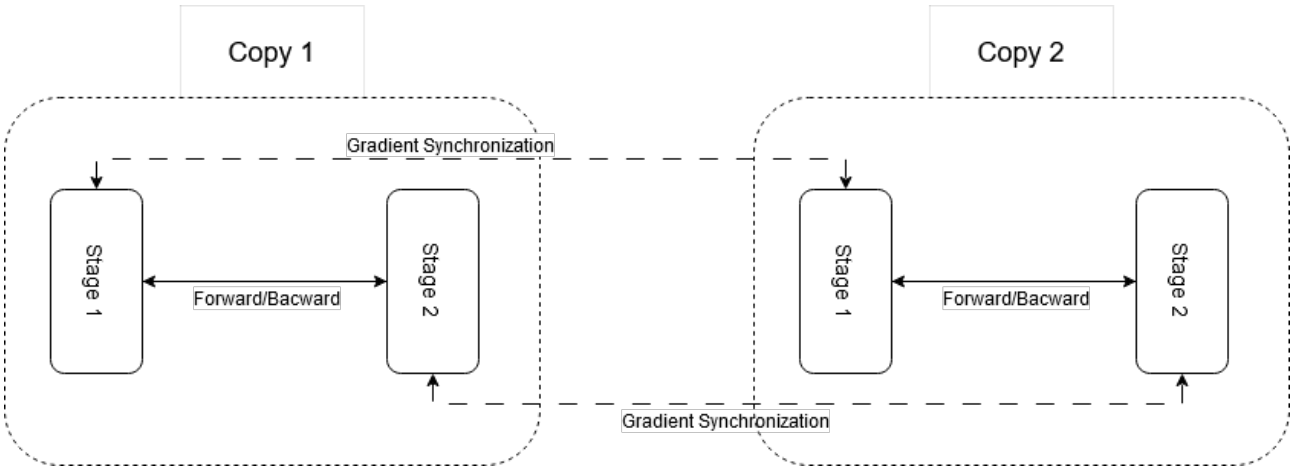

Figure 2: Illustration of the "Data + Pipeline parallelism" approach. Each copy consists of 2 stages, and the gradients are synchronized between corresponding stages (i.e., Stage 1 of Copy 1 with Stage 1 of Copy 2, Stage 2 of Copy 1 with Stage 2 of Copy 2) across the two copies. This ensures proper coordination during the training process.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	25.70	1.18	19.90	26.85
Test	8.86	2.56	7.31	14.09

Table 8: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches, each minibatch split in 10 micro-batches, using the "Data + Pipeline parallelism" approach.

Phase	Mean (s)	Std (s)	Min (s)	Max (s)
Training	19.19	0.24	18.89	20.77
Test	7.34	0.13	7.23	7.66

Table 9: Summary statistics (seconds) obtained from 100 training minibatches and 20 test minibatches, each minibatch split in 5 micro-batches, using the "Data + Pipeline parallelism" approach.

Examining the results, it is clear that they are very similar to those of the 2-stage pipeline, suggesting that the overhead introduced by gradient synchronization is negligible. This indicates that the combination of data and pipeline parallelism allows for a significant improvement in computational performance. By using two copies of the model, each processing only half of the dataset, the system improves efficiency by reducing the amount of data handled by each copy, while maintaining nearly the same processing time per minibatch as the 2-stage pipeline. Given the nearly identical time per minibatch in the case of 5 microbatches, this configuration achieves a speedup of about 2 over the 2-stage pipeline approach (1.94 to be exact) and a speedup of nearly 4 (3.8) over the sequential model.

The use of two copies of the model and the splitting of the data set between them introduces the possibility of slight differences in the calculated loss compared to the sequential model. These differences could be due to factors such as uneven distribution of data features across the splits, slight variations in gradient updates due to synchronization, between the copies, by averaging. Ideally, the loss should not differ significantly from that of the sequential model, as maintaining consistency in the results is critical. Figure 3 illustrates the loss plot, comparing the results of this approach, using 5 microbatches, to those of the sequential model and highlighting any deviations. The training loss curves for the sequential and data parallel (DP) models are nearly identical, indicating that the DP approach achieves the same accuracy as the sequential method. This result highlights that data parallelism (including pipeline parallelism) allows for faster training without sacrificing model performance.

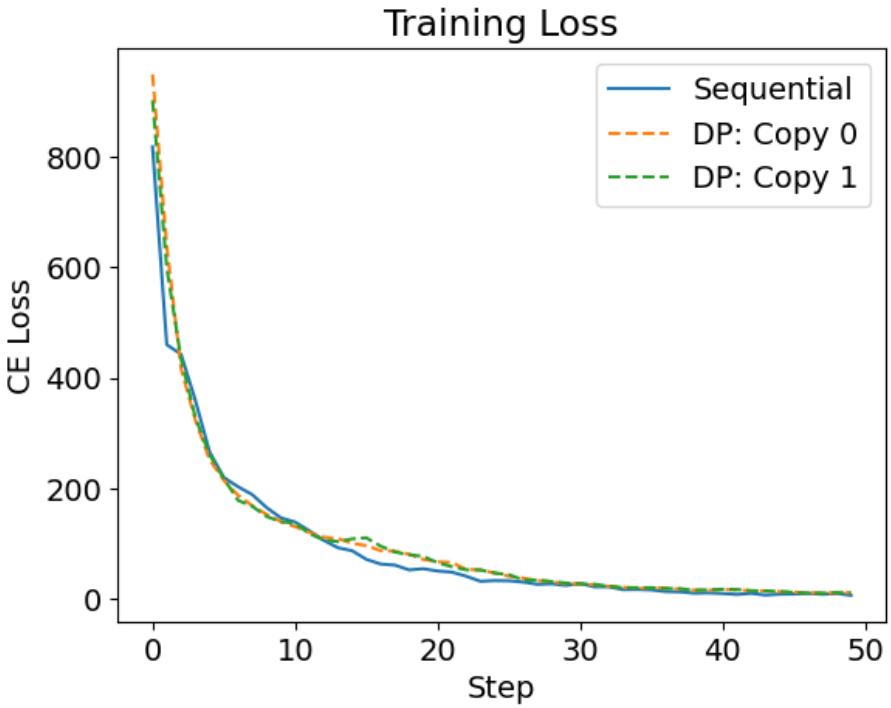

Figure 3: Comparison of Training Loss (Cross Entropy) for Sequential and Data Parallel (DP) Models. The DP approach includes two model copies: Copy 0 and Copy 1.

5 Technical Setup

The computational experiments for this project were performed on a high performance computing (HPC) cluster managed by Slurm. Each node in the cluster is equipped with two Intel Xeon Gold 5120 processors, providing a total of 56 logical CPUs and 125 GiB of memory. The study case model consists of eight ViG blocks with a total of 1,120,410 parameters and 138,258 parameters per block. PyTorch 2.4.1 and PyTorch Geometric (PyG) 2.6.1 were used as deep learning frameworks.

6 Conclusion and Future Work

This study investigated the efficiency of pipeline parallelism and its combination with data parallelism to optimize the computational performance of graph neural networks (GNNs). The single pipeline approach showed good efficiency, with minimal overhead and effective parallelization in the 2-stage configuration. However, scaling to more stages introduced some limitations. The combination of data and pipeline parallelism showed even greater performance improvements. By splitting the data set and using two copies of the model, each processing half of the data, the system maintained similar minibatch processing times while significantly improving efficiency. Minimal gradient synchronization overhead further improved throughput. Both approaches proved effective in improving the training performance of GNNs and offer promising solutions for handling large datasets and efficiently scaling models.

Future work can further explore the proposed approaches by applying them to models of varying complexity to evaluate their scalability and performance on different architectures. In addition, the study of map parallelism for GNNs could uncover new opportunities for performance optimization. Another important direction is to study the throughput of pipeline parallelism during inference. This could involve exporting models to more efficient languages, such as C++, and implementing the pipeline using distributed technologies, such as MPI, to optimize performance during inference. In addition, exploring data parallelism specifically for graph neural networks is critical, especially in cases where the input graph is too large to fit on a single machine. Effective partitioning and distribution of massive graphs is essential for improving scalability, training efficiency, and model effectiveness, as discussed in recent surveys [4] and [1].

References

[1] Maciej Besta and Torsten Hoefler. “Parallel and Distributed Graph Neural Networks: An In-Depth Concurrency Analysis”. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 46.5 (2024), pp. 2584–2606. DOI: 10.1109/TPAMI.2023.3303431. URL: https://doi.org/10.1109/TPAMI.2023.3303431.

[2] Kai Han et al. "Vision GNN: An Image is Worth Graph of Nodes". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 8291–8303. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/3743e69c8e47eb2e6d3afaea80e439fb-Paper-Conference.pdf.

[3] Yanping Huang et al. "GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper_files/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

[4] Yingxia Shao et al. "Distributed Graph Neural Network Training: A Survey". In: *ACM Comput. Surv.* 56.8 (2024), 191:1–191:39. DOI: 10.1145/3648358. URL: https://doi.org/10.1145/3648358.