An analysis of multi-GPU training performance in GNN libraries

Antonia Boca
Department of Computer Science
University of Cambridge
aib36@cam.ac.uk

Abstract

Graph neural networks (GNNs) are becoming increasingly popular, with successful applications ranging from recommender systems to protein prediction for drug design. Due to the intrinsically unstructured nature of graphs, GNNs are not easily parallelised, since adjacency matrices can be highly sparse and the number of neighbours of a node can vary drastically within one graph. Efforts to accelerate GNN training on GPUs have been made by two PyTorch libraries that provide deep learning tools for irregularly structured data: *PyTorch Geometric* and *Deep Learning Library*. While the former uses a *scatter-gather* abstraction for accelerating computation, the latter implements *sampled dense-dense and sampled sparse-dense matrix multiplication* primitives in order to provide node and edge parallelism. As the reported speedups of these frameworks are limited to CPU or single-GPU experiments, this project investigates the scalability of the two frameworks to multi-GPU training. Results indicate that the primary bottleneck in both libraries is data loading; attempts to parallelise graph sampling and loading end up in deadlocks when using more than 2 GPUs, suggesting that distributed training support is still limited.

The code used in this report can be found at: `https://github.com/semiluna/R244`.

1 Introduction

Graph neural networks (GNNs) are a type of neural network that operate on graphs and are able to capture structural information in the data by leveraging the existent relationships between entities. All GNNs use some form of *neural message passing*, where vector messages are exchanged between nodes and updated using a neural network [Gilmer et al., 2017]. Because they recognise and use graph structure, GNNs are difficult to parallelise and prohibitively expensive to train and deploy on large graphs.

PyTorch Geometric[1] [Fey and Lenssen, 2019] and the Deep Graph Library[2] [Wang et al., 2019] are two frameworks built on top of PyTorch [Paszke et al., 2019] that provide common deep learning abstractions for the implementation of GNNs, as well as optimised tensor operations that accelerate the training of such GNNs. The two frameworks use separate graph abstractions to speed up computation, and while each framework claims that their implementation is suited for extensive parallelisation, the authors only report performance metrics for CPU or single-GPU training, leaving multi-GPU training out of the picture.

[1] PyTorch Geometric is abbreviated throughout this project as **PyG**.
[2] Deep Graph Library is abbreviated throughput this project as **DGL**.

Preprint. Under review.

Modern hardware, such as GPUs, utilise multi-threading to achieve high throughput by parallelising massive workloads. As such, multi-GPU training is crucial for large-scale training of performant neural networks, and GNNs are no exception: for example, the Reddit dataset [Hamilton et al., 2017] is a well-known graph dataset that contains 232,965 posts belonging to different communities. Each node has an average degree of 492. Within this context, it is necessary to understand the shortcomings of present GNN frameworks if the research community wishes to continue developing bigger and better models that can be trained in a timely manner. Therefore, this project analyses the performance of **DGL** and **PyG** in a multi-GPU environment, discussing both the advantages and disadvantages of their approaches relative to two standard GNN models.

The rest of this report is structured as follows: Section 2 introduces the GNN models I will use for the experiments; Section 3 presents and compares the libraries I use to implement the GNN models; Section 4 presents the implementation details of the models and presents an overview of how data loading is performed for graphs. Section 5 presents the experimental results and section 6 summaries the main limitations of my approach and some concluding remarks.

2 Graph neural networks

Graph neural networks are characterised by the *neural message passing* framework, in which vector messages are exchanged between nodes and then updated using neural networks. At message passing iteration k, a hidden embedding $\mathbf{h}_u^{(k)}$ corresponds to each node $u \in \mathcal{V}$, and an update happens according to information aggregated from u's graph neighbourhood $\mathcal{N}(u)$, as seen in Equation 2.

$$\mathbf{h}_u^{(k+1)} = \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k+1)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k+1)}, \forall v \in \mathcal{N}(u)\})) \tag{1}$$

$$= \text{UPDATE}^{(k)}(\mathbf{h}_u^{(k+1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}) \tag{2}$$

Note that in general the AGGREGATE function must be permutation-invariant.

2.1 Node classification

One important task in GNNs is that of node classification, where the goal is to predict class labels for some nodes in a graph. For example, in a social network graph, the nodes might represent individuals, and the task could be to predict the occupation of each individual based on their attributes and connections in the graph. Node classification can be further categorised as either *transductive* or *inductive*. In the former setting, the training, validation and test nodes are all on the same graph and the entire graph is made available to the model during training, but the loss is computed only using the training nodes. In the latter setting, some nodes and edges (and even graphs) are completely unseen during training, meaning that the GNN trains only on subgraphs of the original graphs available in the dataset.

2.2 Graph Attention Networks

One popular kind of GNNs were introduced by Veličković et al. [2017], called Graph Attention Networks (GATs). For these networks, the UPDATE and AGGREGATE functions form the following attentional operator:

$$\mathbf{h}_u^{(k+1)} = \alpha_{u,u}\Theta\mathbf{h}_u^{(k)} + \sum_{v \in \mathcal{N}(u)} \alpha_{u,v}\Theta\mathbf{h}_v^{(k)}, \tag{3}$$

where $\alpha_{u,v}$ is an attention coefficient computed as

$$\alpha_{u,v} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T[\Theta\mathbf{x}_u || \Theta\mathbf{x}_v]))}{\sum_{k \in \mathcal{N}(u) \cup \{u\}} \exp(\text{LeakyReLU}(\mathbf{a}^T[\Theta\mathbf{x}_u || \mathbf{x}_k]))}. \tag{4}$$

2.3 GraphSAGE Networks

Hamilton et al. [2017] introduced the GraphSAGE layer, designed specifically for *inductive* learning on large graphs. Message passing layers following the GraphSAGE framework use the following

operators:

$$\mathbf{h}_{\mathcal{N}(u)}^{(k+1)} = \text{aggregate}(\{\mathbf{h}_v^k, \forall v \in \mathcal{N}(u)\}) \tag{5}$$

$$\mathbf{h}_u^{(k+1)} = \sigma(\mathbf{W} \cdot \text{concat}(\mathbf{h}_u^k, \mathbf{h}_{\mathcal{N}(u)}^{(k+1)})) \tag{6}$$

$$\mathbf{h}_u^{(k+1)} = \text{norm}(\mathbf{h}_u^{(k+1)}) \tag{7}$$

This project will perform inductive training on the Reddit dataset [Hamilton et al., 2017] using two GNN models: one GAT and one GraphSAGE network.

3 Deep learning libraries

3.1 Deep Graph Library

Wang et al. [2019] introduced the Deep Graph Library, a GNN library that is framework-neural and can be added on top of PyTorch [Paszke et al., 2019], Tensorflow [Abadi et al., 2015] and MXNet [Chen et al., 2015]. Within DGL, graphs are "first-class citizens" represented by the `DGLGraph` data structure, on which standard message passing layers such as GAT (e.g. `GATConv`) and GraphSAGE (e.g. `SAGEConv`) operate directly. This way, DGL can choose how to optimise matrix multiplication depending on whether the computation is done in the forward pass or in the backward pass.

SpMM and SDDMM. The main objective of DGL is to control sparse tensor storage management and operations. To optimise such operations, it generalises all message passing computations to either a *generalised sampled dense-dense matrix multiplication* (g-SDDMM) or *generalised sparse-dense matrix multiplication* (g-SpMM). Sematincally, the former corresponds to the node-wise computation $\mathbf{Y} = \mathbf{AX}$, for node matrix $\mathbf{X} \in \mathbb{R}^{|\mathcal{V}| \times d}$ and sparse adjacency matrix $\mathbf{A}$, while the latter corresponds to the edge-wise computation $\mathbf{W} = \mathbf{A} \odot (\mathbf{X}\mathbf{X}^T)$.

Wang et al. [2019] prove that the gradient of the loss function w.r.t. to g-SDDMM and g-SpMM inputs can be expressed in terms of other g-SDDMM and g-SpMM operations. This formulation enables fused computation, therefore avoiding the generation of intermediate storage for messages or the copying of node representations to edges; this strategy therefore benefits memory consumption and brings a speed up to the message-passing layers.

Parallelisation strategies. *Node* and *edge parallelism* are two common strategies for taking advantage of GPU multi-threading when training GNNs. With node parallelism, each thread is in charge of the adjacency list of a node, while with edge parallelism, each thread is in charge of one edge. Each of these strategies depends on the computation pattern being used, how the adjacency matrix is stored and how many edges or nodes are in the graph. DGL uses a node parallel strategy for g-SpMM computation and an edge parallel one for g-SDMM; however, this is just a heuristic.

3.2 PyTorch Geometric

PyTorch Geometric [Fey and Lenssen, 2019] is another deep learning library providing the necessary graph abstractions to be used when building GNNs. In contrast to DGL, PyG accelerates sparse operations across the GNN model by implementing a *scatter/gatter* interface with dedicated CUDA kernels. Node features are scattered to their neighbours and then gathered to perform aggregation. This interface was inspired by `pytorch-scatter`,[3] a PyTorch extension that support sparse tensor operations. This way, PyG avoids the need of performing matrix multiplication, and instead uses adjacency lists to represent edges in a graph.

The *scatter/gather* interface alternates between node parallel and edge parallel computation; the authors argue that this is advantageous for graphs containing nodes with few neighbours. Indeed, their evaluation compares PyG to DGL on the Cora [McCallum et al., 2000], Citeseer Giles et al. [1998] and Pubmed Sen et al. [2008] datasets; these three datasets are well-known in the research community, but contain small graphs that can be easily trained on a single CPU.

[3]The library can be found at `https://pytorch-scatter.readthedocs.io/en/1.3.0/index.html`.

3.3 PyTorch Lightning and DDP strategies

PyTorch Lightning[4] is a research framework built on top of PyTorch that abstracts away common boilerplate code of neural networks. The framework automatically detects available GPUs and modifies training such that it runs in a distributed manner. This project uses PyTorch Lightning on top of DGL and PyG for automated parallelisation in a multi-GPU environment.

PyTorch's `DistributedDataParallel`. One of the parallelisation strategies used by PyTorch Lightning is `DistributedDataParallel (DDP)`. This PyTorch container parallelises the application by splitting the input across the specified devices by chunking in the batch dimension. Using the `DDP` container by itself requires a lot of boilerplate code, but it can be abstracted away by Lightning; the framework orchestrates the following procedure:

- A process will be created on each GPU (and across each node, if there are multiple nodes).
- Each GPU will receive a subset of the overall dataset.
- Each process within each GPU will initialise the model.
- Each process will perform a forward and backward pass.
- The gradients computed by each process are synced and averaged across all processes.
- Each process updates its optimiser.

4 Implementation

I implement two types of graph neural networks: a GraphSAGE model [Hamilton et al., 2017] and a GAT model [Veličković et al., 2017]. The hyperparameters of the two network architectures are summarised in Table 1.

Table 1: Sumamry of architectures used.

Model	Layers	Hidden features	Aggregation	Attention heads	Dropout
GraphSAGE	3	256, 256, 256	mean	-	0.5, 0.5, 0.5
GAT	3	16, 16, 16	-	1, 1, 1	0.6, 0.6, 0.6

The two models were wrapped around a Pytorch Lightning `LightningModule`. Both models are trained with the Adam optimiser [Kingma and Ba, 2014] with a learning rate of $5e^{-3}$ and a weight decay of $5e^{-4}$.

4.1 Data loading

The models are trained on the Reddit dataset, introduced by Hamilton et al. [2017]. It contains 232,965 posts belonging to different communities. Each node has an average degree of 492. When loaded in memory, this dataset can occupy up to 6GB, so it is unfeasible to train the models on transductive tasks for such a graph. A standard way of dealing with this problem is to perform *neighbour sampling*. This strategy refers to the process of selecting a subset of nodes and their corresponding edges from the graph to be used as input for a given iteration of training. This subset, often referred to as a "mini-batch," is used to update the model's parameters. The specific nodes and edges that are selected can be chosen randomly or according to some predefined criterion. Both PyG and DGL have dedicated neighbour samplers that can be used in conjunction with PyTorch's `DataLoader`.

PyTorch's `DataLoaders`. In PyTorch, a `DataLoader` is used to load and preprocess data for use in training or testing a machine learning model. A key feature of the DataLoader is the use of multiple worker processes, known as *data workers*, to parallelise the loading and preprocessing of data. When a `DataLoader` is created, we can specify the number of data worker processes to use.

[4] `https://www.pytorchlightning.ai`

Each data worker runs in its own process and is responsible for loading and preprocessing a portion of the data. The data workers run in parallel and communicate with the main process using Python's `multiprocessing` module. This allows the data loading and preprocessing to be done in parallel, which can significantly speed up the overall training.

4.2 `DataLoaders` and deadlock

While running experiments, I encountered deadlock for all runs that were configured to use 4 GPUs and **more than one data worker**. While I do not have a clear explanation for this behaviour, searching through multiple open Github issues seems to suggest that PyTorch's `DataLoader` used with multiple worker processes will deadlock when the *data sampling* algorithm is too slow; since both PyG and DGL implement their own sampling procedures in order to perform *neighbour sampling*, I believe the deadlock is generated by a mismatch between the two GNN libraries and PyTorch's parallelisation strategy. As a consequence, all my experiments configured with 4 GPUs use a single data worker. The impact of this change will be discussed in Section 5.

5 Experiments

I ran all experiments on a Google Cloud **N1** machine with 16GB of RAM. To this instance I attached 1, 2 or 4 NVIDIA Tesla T4 GPUs. I train each model for 50 epochs, record the training time on each GPU and report the longest time it took any of the subprocesses to finish training. Table 2 summarises the results for all possible configurations of model, framework and batch size.

Table 2: Longest training time of any training subprocess for the two GNN models implemented either using PyG or DGL. The batch size is **per training process**. The lower training time between PyG and DGL for a specific configuration is in **bold**. For an explanation of the number of data workers, please refer to Section 4.2.

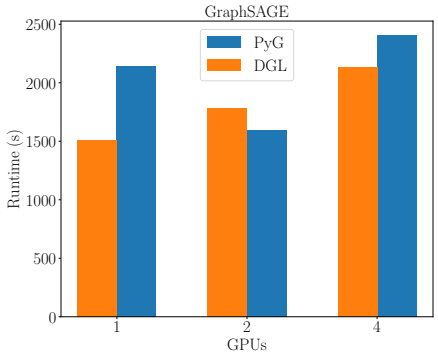
Model	GPUs	Data Workers	Running time (s)					
			Batch size = 512		Batch size = 1024		Batch size = 2048	
			PyG	DGL	PyG	DGL	PyG	DGL
GraphSAGE	1	8	4758	**3789**	3168	**2342**	2138	**1511**
	2	8	**2818**	4369	**2119**	2866	**1593**	1786
	4	1	**4306**	5932	**3221**	3549	2407	**2130**
GAT	1	8	**3270**	3536	**2096**	2272	1550	**1504**
	2	8	**2624**	4108	**1892**	2680	**1383**	1727
	4	1	**3946**	5976	**2980**	3453	2211	**2054**

5.1 Discussion

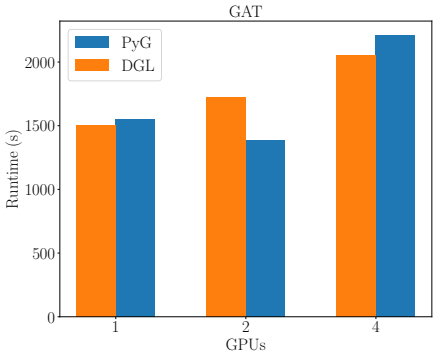
The results in Table 2 show that doubling the number of GPUs does not halve the training runtime; on the contrary, as discussed in 4.2, using a large number of GPUs will result in a data loading *bottleneck*. Neither PyG nor DGL have implemented highly parallisable neighbour sampling algorithms, thus the main limitation of these libraries is their ability to accelerate graph sampling and loading.

PyG is better when using smaller batch sizes. We can see that PyG outperforms DGL when using smaller batch sizes. As explained in Section 4.1, neighbour sampling returns a subgraph of the original graph as a training batch. When the training batch is smaller, the subgraph itself will be smaller; under this assumption, PyG's scatter-gather approach performs better.

DGL is better in single-GPU environments. Table 2 shows that in 4 out of 6 single-GPU experiments, DGL outperforms PyG. Since the mini-batches themselves cannot contain large graphs, and therefore, as explained above, PyG should outperform DGL, these results indicate that DGL's neighbour sampling algorithm is faster in a single-GPU environment. However, the fact that DGL's runtime *increases* when using 2 GPUs is a strong indication that data loading cannot be parallelised



(a) Training runtime of PyG and DGL on the GraphSAGE model with a batch size of 2048.



(b) Training runtime of PyG and DGL on the GAT model with a batch size of 2048.

Figure 1: Training runtime trends when increasing the number of GPUs.

.

with DGL. This is in contrast with PyG, for which adding a second GPUs results in a 70% speed up. This behaviour can be better visualised in Figure 1.

6 Conclusion

This project investigated the behaviours of two popular GNN libraries for PyTorch when training using multiple GPUs. Two commonly used GNN models, GAT and GraphSAGE, were trained to perform node classification using neighbour sampling on the Reddit dataset. Experiments were run on Google Cloud with 1, 2 and 4 GPUs, and results indicate that the two libraries are not well suited for multi-GPU training due to data loading bottlenecks. Maybe most striking is the fact that trying to parallelise data loading to multiple subprocesses results in deadlocks when using 4 GPUs, irrespective of the framework used.

Data loading aside, this project cannot indicate to a clear "winner" when it comes to GNN training; as we have seen, DGL performs better in single-GPU environments, and I believe this is due to its optimised neighbour sampling algorithm. However, adding multiple GPUs *hurts* performance, indicating that distributed training is not properly addressed in DGL. Alternatively, PyG peforms better with small batch sizes and it is better suited for distributed training; as we have seen, adding a second GPU can reduce training time by up to 70%.

While this analysis can be used as a starting point for improving the performance of distributed training in GNN libraries, there are a couple of limitations that I wish to address.

Limited scope. Due to limited time and resources, this project has a limited scope along two dimensions: firstly, it trains only two types of GNN networks. While GraphSAGE and GAT are commonly used in the research community, other models, such as GIN [Xu et al., 2018] and GCN [Kipf and Welling, 2016], may require different parallelisation strategies that are worth investigating. Secondly, GNN frameworks provide support for many more training tasks other than node classification, such as link preddiction and graph classification. Moreover, these tasks can further be divided into transductive and inductive tasks, requiring different types of optimisations. As such, I believe it is important to compare the full range of task support the two libraries have to offer.

Choice of deep learning libraries. My analysis focuses on the difference between the Deep Graph Library and PyTorch Geometric as extensions to PyTorch. However, Deep Graph Library can run on top of other deep learning libraries, such as Tensorflow [Abadi et al., 2015] and MXNet [Chen et al., 2015]. In fact, Tensorflow is presently developing a new GNN library, TF-GNN [Ferludin et al., 2022], that is supposed to provide better and faster GNN support. Another GNN library that provides Tensorflow support is Spektral [Grattarola and Alippi, 2020], whose performance is also worth investigating.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org.

Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015. URL `https://arxiv.org/abs/1512.01274`.

Oleksandr Ferludin, Arno Eigenwillig, Martin Blais, Dustin Zelle, Jan Pfeifer, Alvaro Sanchez-Gonzalez, Sibon Li, Sami Abu-El-Haija, Peter Battaglia, Neslihan Bulut, Jonathan Halcrow, Filipe Miguel Gonçalves de Almeida, Silvio Lattanzi, André Linhares, Brandon Mayer, Vahab Mirrokni, John Palowitch, Mihir Paradkar, Jennifer She, Anton Tsitsulin, Kevin Villela, Lisa Wang, David Wong, and Bryan Perozzi. TF-GNN: graph neural networks in tensorflow. *CoRR*, abs/2207.03522, 2022. URL `http://arxiv.org/abs/2207.03522`.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019. URL `https://arxiv.org/abs/1903.02428`.

C. Lee Giles, Kurt D. Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the Third ACM Conference on Digital Libraries*, DL '98, page 89–98, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919653. doi: 10.1145/276675.276685. URL `https://doi.org/10.1145/276675.276685`.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral, 2020. URL `https://arxiv.org/abs/2006.12138`.

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2017. URL `https://arxiv.org/abs/1706.02216`.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL `https://arxiv.org/abs/1412.6980`.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *Information Retrieval*, 3(2):127–163, 2000.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. Deep graph library: Towards efficient and scalable deep learning on graphs. *CoRR*, abs/1909.01315, 2019. URL `http://arxiv.org/abs/1909.01315`.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2018. URL https://arxiv.org/abs/1810.00826.