Decentralized training of graph convolutional networks

Gabe Mancino-Ball
mancig@rpi.com
Department of Mathematical Sciences
Troy, New York, USA

ABSTRACT

Decentralized optimization has received much attention in the recent years for its scalability to large datasets. Additionally, classification problems involving graph structured data have gained much attention for their applicability to recommendation systems and search systems [27]. One such method for solving the graph structured classification problem is the popular graph convolutional network discussed in [9]. Traditionally, this method is housed on one local machine (typically a CPU) for both training and inference. As the size of graph structured data grows, it becomes necessary to distribute the data across multiple computing nodes/GPUs or even multiple data centers. This distribution of data invites the use of decentralized optimization techniques to aid in solving the underlying classification problems. The goal of this work is to empirically analyze the effect that decentralized training of graph convolutional networks has on relevant performance metrics such as training time and test-set accuracy. This work provides a novel `PyThon` framework that utilizes `mpi4py` to perform the training and inference of graph convolutional networks in a decentralized manner.

KEYWORDS

decentralized optimization, graph neural networks, distributed computing

ACM Reference Format:
Gabe Mancino-Ball. 2021. Decentralized training of graph convolutional networks. In *Parallel Computing: Spring 2021.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Classification problems in machine learning can range from classifying images [10], to words [18], to sounds [4]. Recently, works have focused on *graph* structured data for *node* classification problems. In these problems, datasets are represented as graphs where each point in the dataset contains a feature vector $\mathbf{x}_j \in \mathbb{R}^p$ and a label $y_j \in \{0, \dots, C\}$ with $j = 1, \dots, m$. Here, C represents the number of possible classes a node can belong to. Datasets are structured such that the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ representing how the data points are interconnected, can be expressed via an *adjacency matrix* $\mathbf{A} \in \mathbb{R}^{m \times m}$. The challenging aspect of this problem is that there exists only a subset of available data with known labels to use as training data. We index this training set as $\mathcal{T} \subset \{1, \dots, m\}$, where it is assumed that $|\mathcal{T}| << m$. Examples of such datasets include social networks and knowledge graphs [27]. Traditionally, each graph is stored on a device (CPU in the traditional case) and trained with the Adam [8] optimizer. As graphs become larger and larger, indeed pushing upwards of 1 billion plus nodes [27], distributing the data among many computing devices[1] becomes not only desirable, but necessary. The goal of this work is to study the effects that applying methods from *decentralized optimization*, has on solving node classification problems.

Consider a network of N "agents" (computing devices, GPUs, etc.) interconnected over a graph $\mathcal{G}_{\text{comm}} = (\mathcal{V}_{\text{comm}}, \mathcal{E}_{\text{comm}})$, where $\mathcal{V}_{\text{comm}} = \{1, \dots, N\}$ represents the agents and $\mathcal{E}_{\text{comm}}$ details the connection links between said agents. Decentralized optimization aims to solve the following problem:

$$\min_{\boldsymbol{\theta}} f(\boldsymbol{\theta}) \triangleq \frac{1}{N} \sum_{i=1}^{N} f_i(\boldsymbol{\theta}), \tag{1}$$

where each $f_i : \mathbb{R}^p \to \mathbb{R}$ is a local, non-convex loss function known only to agent i. In this paper, each f_i represents a loss function applied to a graph neural network and $\boldsymbol{\theta}$ represents the parameters of the network. To solve (1) by a first order method (e.g. gradient descent), each agent i in the network would need access to every other ∇f_j, for all $j \neq i$, at every iteration. In decentralized optimization, it is assumed that agent i only has access to f_i and ∇f_i, to aid in preserving privacy within the network [14]. Additionally, (1) does not encode any information about the underlying agent connectivity structure encoded in the graph's edges, $\mathcal{E}_{\text{comm}}$. To encompass these necessary considerations, problem (1) can be reformulated as

$$\min_{\boldsymbol{\theta}_1, \dots, \boldsymbol{\theta}_N} \frac{1}{N} \sum_{i=1}^{N} f_i(\boldsymbol{\theta}_i), \quad \text{subject to} \quad \boldsymbol{\theta}_i = \boldsymbol{\theta}_j, \forall i, j \in \{1, \dots, N\}. \tag{2}$$

This reformulation allows each agent to have its own copy of the variable $\boldsymbol{\theta}$, which it can use to compute local gradients in a private manner. The equality constraint $\boldsymbol{\theta}_i = \boldsymbol{\theta}_j$ enforces consensus among the local parameters. It is easy to verify that problems (1) and (2) have the same minima. Many methods have been developed recently to solve (2), however, this work is focused on applying one such decentralized method to train a graph neural network for the purpose of node classification. The remainder of this work is structured as follows:

[1] We consider a computing device in this work to be a single GPU. Other computing devices could be CPUs, cell phones, etc. [14].

Parallel Computing , Spring 2021, RPI

ACM ISBN XXX-XXXX-XXXX…$00.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- a brief literature review is outlined, with emphasis on decentralized optimization methods and graph neural network architectures,
- the graph convolutional network architecture is discussed, as well as the process which this work takes to distribute graph data among GPUs,
- the method of choice for solving (2) is discussed, with a statement of the theoretical results, and finally
- numerical results are presented on the node classification problem and implementation details are presented; we end with conclusion statements and goals of future work.

1.1 Literature review

1.1.1 Decentralized optimization. Decentralized optimization dates back decades to the seminal work of [2]. In 2009, Nedić and Ozdaglar proposed a decentralized subgradient method for solving (1), under the assumption that each f_i was convex, but potentially non-differentiable [15]. Following this work, several other first order methods have been proposed and analyzed. The classic **D**ecentralized **G**radient **D**escent, or **DGD**, algorithm was shown to converge in the convex setting [23] and later its convergence was generalized to the non-convex setting [24]. This work will focus on how DGD can be used in solving node classification problems. For convex problems, other notable first order methods for solving (2) are EXTRA [20], ADMM [19], and IDEAL [1]. These methods all require that each f_i has L-Lipschitz continuous gradient (see (10) for a definition), while IDEAL further requires that f_i is μ-strongly convex[2].

This work is primarily concerned with methods that have provable convergence guarantees for non-convex (2). Prominent methods for solving non-convex (2) include Prox-PDA [6] and NEXT [13]. While NEXT actually lacks any convergence rate results, it does exhibit asymptotic convergence guarantees. Prox-PDA exhibits a sublinear convergence rate on the order $O\left(\frac{1}{K}\right)$ where K is the total iteration number. Both of these methods rely on creating a strongly convex subproblem to be solved at every iteration. NEXT creates a generic *strongly convex approximation* and Prox-PDA relies on adding a penalty term to the local objective function. The major difference between these two methods is that NEXT utilizes gradient tracking and Prox-PDA maintains a dual variable. It is worth exploring how these methods compare to DGD on the node classification problem, however, that comparison remains outside of the scope of this work.

Other solvers for non-convex (2) include stochastic methods such as DSGT [25], D-GET [21], and D-PSGD [11]. Stochastic methods remain important for many machine learning applications as they encourage generalizable parameters that perform well on unforeseen data, but for the node classification problem considered in this work, the small training set size may not be compatible with stochastic methods [9]. Asynchronous methods such as AD-PSGD [12] and APPG [26] consider an update rule that removes the need to synchronize the agents at every step. Push-Pull [16] operates in a time-varying framework where the connections between agents vary at time-step. Both asynchronicity and time-vary network topologies lie outside of the scope of this work and are slated for future considerations.

1.1.2 Graph neural networks. Many neural network architectures have been proposed for solving the node classification problem and while this work only considers the use of **G**raph **C**onvolutional **N**etworks (**GCN**s) [9], it is worth mentioning the other architectures here. Hamilton, et. al. presented the GraphSAGE-GCN architecture for node classification problems [5]. GraphSAGE-GCN is different from GCN in that it does not use the graph Laplacian of the underlying graph structure. The Graph Isomorphism Network (GIN) can also be adapted to the node classification problem [22]. Like GraphSAGE-GCN, GIN does not use the Laplacian of the underlying graph structure; instead, the GIN *learns* an importance weighting for each node's information (each agent learns its own weighting; neighbor weighting remains constant). While both GraphSAGE-GCN and GIN have shown to perform well in practice, we remain focused on utilizing the GCN as proposed in [9]. Our work is closely related to [17], where the authors have proposed a variation of DGD that they applied to the node classification problem.

1.2 Notation

We use bold face capital letters and bold face lowercase letters, e.g. $\mathbf{X}$ and $\mathbf{x}$, to denote matrices and vectors, respectively. Denote the element in the i^{th}-row and j^{th}-column of a matrix $\mathbf{X} \in \mathbb{R}^{N\times p}$ to be x_{ij}. Denote the Frobenius norm of a matrix to be $\|\cdot\|_F$ and the Euclidean norm of a vector to be $\|\cdot\|$. The spectral radius of an $N \times N$ matrix $\mathbf{X}$ is given by $\lambda(\mathbf{X})$, with a general ordering of $\lambda_1(\mathbf{X}) \geq \lambda_2(\mathbf{X}) \geq \cdots \geq \lambda_N(\mathbf{X})$, where $\lambda_i(\mathbf{X})$ is the i^{th} largest eigenvalue of $\mathbf{X}$. The q^{th} power of a diagonal matrix, $\mathbf{D}^q \in \mathbb{R}^{N\times N}$ is given by d_{ii}^q for all $i = 1, \ldots, N$ and 0 for all other entries. For node j in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, define the set of neighbors of j to be $\mathcal{N}(j) \triangleq \{i \in \mathcal{V} : (j, i) \in \mathcal{E}\}$.

2 GRAPH CONVOLUTIONAL NETWORKS

To specify the problem structure of (2), let $\{(\mathbf{x}_j, y_j)\}_{j=1}^m$ be a set of feature vectors ($\mathbf{x}_d \in \mathbb{R}^p$) and labels ($y_d$, where $y_d \in \{1, \ldots, C\}$), which will in general be referred to as the *data*. The data is assumed to be structured in such a way that a graph exists to model how the data is connected - for this work, we assume the graph is undirected and we will refer to each data point pair, $(\mathbf{x}_d, y_d)$ as being stored on a *node*. Suppose that every agent in the network contains a subset of said data (in practice, this will involve *partitioning* the graph structure, which will be discussed in the next section) so that agent i holds data $\{(\mathbf{x}_d, y_d)\}_{d\in\mathcal{D}_i}$ and $\cup_i \mathcal{D}_i = \{1, \ldots, m\}$. Additionally, assume that $\mathcal{T}_i \triangleq \mathcal{T} \cap \mathcal{D}_i$ is the set of data with known labels located on agent i. Then, each local loss function is of the form

$$f_i(\boldsymbol{\theta}_i) \triangleq \sum_{d\in\mathcal{T}_i} \ell(\mathbf{x}_d, y_d; \boldsymbol{\theta}_i) \tag{3}$$

where $\ell(\mathbf{x}_d, y_d; \boldsymbol{\theta}_i)$ is the *negative log-likelihood loss* function given by

$$\ell(\mathbf{x}_d, y_d; \boldsymbol{\theta}_i) \triangleq -\sum_{c=1}^{C} \mathbb{I}[y_d = c] \cdot \log(g(\boldsymbol{\theta}_i; \mathbf{x}_d)_c) \tag{4}$$

[2] A differentiable function $g : \mathbb{R}^p \to \mathbb{R}$ is called $\mu-$strongly convex if there exists $\mu > 0$ such that for all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^p$, $g(\mathbf{y}) \geq g(\mathbf{x}) + \nabla g(\mathbf{x})^\top(\mathbf{y} - \mathbf{x}) + \frac{\mu}{2}\|\mathbf{x} - \mathbf{y}\|^2$.

where $\mathbb{I}[y_d = c] = \begin{cases} 1 & \text{if } y_d = c \\ 0 & \text{otherwise} \end{cases}$ is the class indicator function of the label and $g(\boldsymbol{\theta}_i; \mathbf{x}_d)_c$ is the c^{th} output of a neural network, $g : \mathbb{R}^p \to \mathbb{R}^C$ (with architecture to be defined shortly), parameterized by $\boldsymbol{\theta}_i$. Under this formulation, each agent requires the data $\{(\mathbf{x}_d, y_d)\}_{d \in \mathcal{D}_i}$, and access to the function $g(\cdot;\cdot)$ and its gradients, which we denote $\nabla g(\cdot;\cdot) \triangleq \nabla_{\boldsymbol{\theta}} g(\cdot;\cdot)$.

For the moment, assume that $N = 1$ so that all of the data lies on one GPU. Then the *adjacency matrix* of the underlying graph structure of the data is given by $\mathbf{A} \in \mathbb{R}^{m \times m}$ such that $a_{ij} = 1$ if nodes i and j are connected and 0 otherwise, and the *degree matrix* is a diagonal matrix $\mathbf{D} \in \mathbb{R}^{m \times m}$ such that $d_{ii} = \sum_{j=1}^{m} a_{ij}$ for all $i = 1, \dots, m$. Utilizing the **renormalization trick** [9], define the following matrix:

$$\hat{\mathbf{A}} \triangleq \tilde{\mathbf{D}}^{-1/2} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-1/2} \tag{5}$$

where $\tilde{\mathbf{A}} \triangleq \mathbf{A} + \mathbf{I}$ is self-loop version of the adjacency matrix and $\tilde{\mathbf{D}}$ is the corresponding degree matrix of $\tilde{\mathbf{A}}$. From (5), a neighbor communication (i.e. local sharing) of data can be computed as $\hat{\mathbf{A}}\mathbf{X}$ where the j^{th} row of $\mathbf{X}$ corresponds to $\mathbf{x}_j$. Using this matrix multiplication, we define the neural network architecture used in this work to be,

$$g(\boldsymbol{\theta}_i; \mathbf{X}) \triangleq \sigma\left(\hat{\mathbf{A}}\left[\hat{\mathbf{A}}\mathbf{X}\boldsymbol{\theta}_i^{(1)}\right]_+ \boldsymbol{\theta}_i^{(2)}\right) \tag{6}$$

where σ is the softmax function and $[\cdot]_+$ is the rectified linear unit (ReLU) activation function. Notice that $\hat{\mathbf{A}}$ is not specified as an input to the function; this is because $\hat{\mathbf{A}}$ is assumed to be given for the underlying network topology, thus it is constant for a given set of data, $\mathbf{X}$. Additionally, dropout is applied to the input matrix as well as after the ReLU activation. In [9], the authors note that a general form of a GCN can be given by a stacking of layers of the following form:

$$\mathbf{Z}^{(l+1)} = \sigma_{(l+1)}\left(\hat{\mathbf{A}}\mathbf{Z}^{(l)}\boldsymbol{\theta}_i^{(l+1)}\right) \tag{7}$$

where in general it is assumed that $\boldsymbol{\theta}_i = \begin{bmatrix} \boldsymbol{\theta}_i^{(1)} & \boldsymbol{\theta}_i^{(2)} & \dots & \boldsymbol{\theta}_i^{(\mathcal{L})} \end{bmatrix}$ for an $\mathcal{L}$-layered GCN and $\sigma_{(l+1)}(\cdot)$ is a given activation function for layer $(l+1)$, e.g. ReLU, softmax, etc. Each $\boldsymbol{\theta}_i^{(l)} \in \mathbb{R}^{h_{l-1} \times h_l}$ with $h_1 = p, h_{\mathcal{L}} = C$, and h_l is the dimension of hidden layer l. The GCN in (6) is a 2-layer GCN where each layer is of the form (7); this is because the authors in [9] claim a 2-layer GCN obtains the best performance results in terms of testing accuracy; we do not investigate the validity of this claim in this work.

Notice that (6) has two left multiplications of $\hat{\mathbf{A}}$. Recall that $\hat{\mathbf{A}}$ represents a single-hop communication among nodes in a graph, further, $\hat{\mathbf{A}}$ is doubly stochastic (i.e. $\sum_{j=1}^{m} \hat{a}_{ij} = \sum_{j \neq i} \left(\frac{a_{ij}}{(\sum_{j=1}^{m} a_{ij})+1}\right) + \frac{a_{ii}+1}{(\sum_{j=1}^{m} a_{ij})+1} = 1$ for all $i = 1, \dots, m$ and $\hat{\mathbf{A}}^\top = \hat{\mathbf{A}}$), which means $\hat{\mathbf{A}}\hat{\mathbf{A}}$ is also doubly stochastic and thus corresponds to a 2-hop neighbor communication among nodes. This is summarized in the following proposition.

Proposition 2.1. *Let $\boldsymbol{W} \in \mathbb{R}^{m \times m}$ represent a given communication topology summarized in the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of communication edges, such that $(r, s) \in \mathcal{E}$ if and only if node s is able to communicate with node r. If $\boldsymbol{W}$ is doubly stochastic, meaning $\boldsymbol{We} = \boldsymbol{e}$ and $\boldsymbol{W}^\top = \boldsymbol{W}$, where $\boldsymbol{e} = \begin{bmatrix} 1 & 1 & \dots & 1 \end{bmatrix}^\top \in \mathbb{R}^m$, then the product $\boldsymbol{WW}$ corresponds to a 2-hop neighbor communication for each node in $\mathcal{V}$.*

Proof. Without loss of generality, assume that $w_{ij} > 0$ for all $i, j = 1, \dots, m$. Define $\mathbf{V} \triangleq \mathbf{WW}$ and compute

$$v_{ij} = \sum_{k=1}^{m} w_{ik} w_{kj}.$$

It is clear that v_{ij} represents a communication between nodes i and j. Note that $w_{ik'} w_{k'j} > 0$ if and only if $(i, k') \in \mathcal{E}$ **and** $(k', j) \in \mathcal{E}$ for some $k' \in \{1, \dots, m\}$, thus $v_{ij} > 0$ if and only if there exists a node connecting nodes i and j. □

In general, proposition 2.1 can be extended to k-hop neighbors by computing the product $\mathbf{W}^k$, but we remain focused on the 2-hop scenario in light of (6). Thus, each node will need access to its 2-hop neighbor's information in performing a forward pass of (6); a visual example of this can be found in Figure 1, where it is assumed that the orange data point is the point of interest.

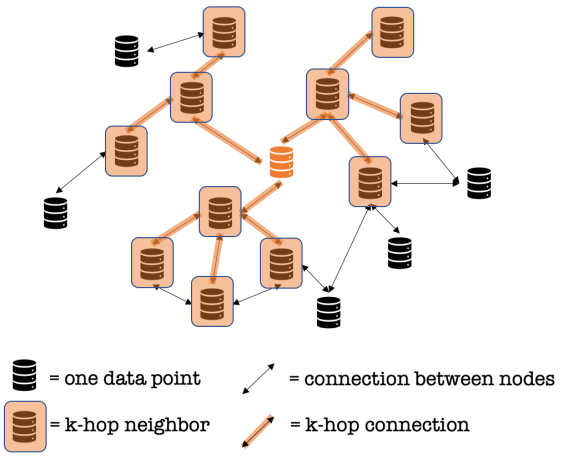

Figure 1: k-hop neighbor example where k=2.

Now that the GCN architecture has been specified by (6), for the case when $N = 1$ in (2), we next discuss how to distribute graph data among $N > 1$ agents so that (2) can be solved using decentralized optimization methods.

2.1 Partitioning with PyMetis

As discussed in the previous section, when $N > 1$, one must find a way to distribute the original graph data across multiple GPUs. In practice, this can be accomplished by *partitioning* the data so that each agent in the network receives a subset of the original dataset. When partitioning the graph, edges will be cut from the original graph - dealing with these edge cuts is non-trivial and actually requires duplication of some edges [27]. Following the example of [27], we use *halo nodes* to handle these edge cuts; before defining a halo node, we build some intuition as to why partitioning of a graph dataset is non-trivial.

Proposition 2.1 and the preceding discussion make note of the fact that each training data point must have access to its 2-hop neighbor for the forward pass of (6), and hence its backward pass in computing the gradients. This means that when we partition a graph, we must account for the fact that some edges cut during the partitioning need to actually be included in the data sent to each agent. Unlike image or sound data, where each point in the dataset has no relation to any other point in the dataset (apart from the assumption that data points are "similar" somehow), graph data actually relies on the connections between data points, meaning that we cannot arbitrarily choose which data points agents receive.

As a small example to demonstrate the importance of preserving the connections between data points, consider the following graph structured dataset:

$$(\mathbf{x}_1, y_1) \leftrightarrow (\mathbf{x}_2, y_2) \leftrightarrow (\mathbf{x}_3, y_3) \leftrightarrow (\mathbf{x}_4, y_4).$$

Assume that $N = 2$ and agent 1 receives $\{(\mathbf{x}_1, y_1), (\mathbf{x}_3, y_3)\}$ and agent 2 receives $\{(\mathbf{x}_2, y_2), (\mathbf{x}_4, y_4)\}$. Then $\hat{\mathbf{A}}_1 = \hat{\mathbf{A}}_2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ so (6) always evaluates to $\frac{1}{2}$ since $e^0 = 1$ (note: we assume that after partitioning the data, $\hat{\mathbf{A}}$ and the data on each agent can be re-indexed to minimize storage requirements; this is why $\hat{\mathbf{A}}_1, \hat{\mathbf{A}}_2 \in \mathbb{R}^{2\times 2}$ here and not in $\mathbb{R}^{4\times 4}$) . Thus, when partitioning the data across GPUs, each GPU **must** receive a cluster of neighboring data points, as opposed to arbitrarily chosen subsets of data points. Under this assumption, agent datasets are now affected by which edges are cut during the initial partitioning. To account for this, *halo nodes* are used. A halo node is a node that is connected to a cluster of data via an edge that will be cut during partitioning. A visual example of partitioning graph data among 4 agents is given in Figure 2. The resulting agent data sets, with halo nodes included, are given in Figure 3.

To facilitate the partitioning of a graph dataset among N nodes, PyMetis is used. PyMetis is a graph partitioning software built in `PyThon` that is based on the original Metis software [7] which was built in `C`. PyMetis offers users the ability to use Numpy arrays when partitioning a dataset, so it is compatible with PyTorch datasets via the `.numpy()` method of a tensor. The main method that PyMetis utilizes is the

```
.part_graph(number_graphs,adjacency_list)
```

method where the `number_graphs` argument is the desired number of output graphs (i.e. N) and `adjacency_list` is a list of Numpy arrays where the i^{th} element in the list contains an array of indices indicating which nodes are connected to node i. PyMetis forms partitions that minimize the total amount of edges cut; this is helpful for us as the more edges cut, the more halo nodes are required for each agent's dataset. After partitioning of the graph data has been completed, the agents must solve (2) by a decentralized method. We discuss our method of choice here.

3 NON-CONVEX DECENTRALIZED GRADIENT DESCENT

DGD is a popular first order method for solving (2) because of its lightweight computation and communication requirements. At the k^{th} iterate in the DGD algorithm, each agent will compute one

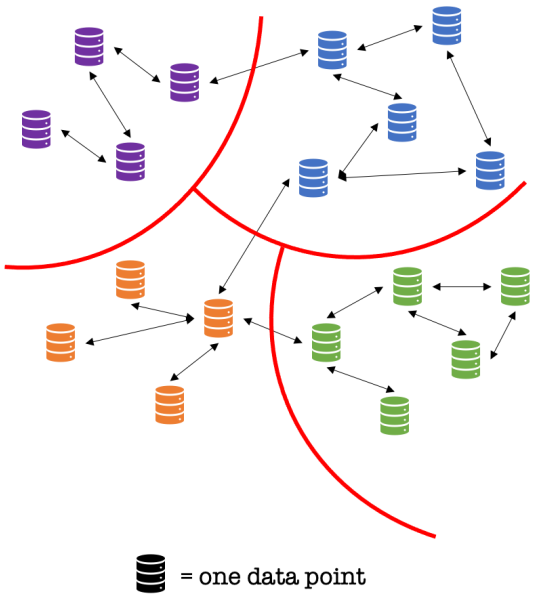

Figure 2: Example of partition cuts on a graph structured dataset. Each red line is a cut.

Figure 3: Resulting agent datasets from the cuts in Figure 2. Nodes with different color than the majority are the halo nodes.

local gradient, $\nabla f_i(\boldsymbol{\theta}_i^k)$, and perform one neighbor communication with all $j \in \mathcal{N}(i)$. To follow the notation of [23, 24], let $\mathbf{W} \in \mathbb{R}^{N\times N}$ represent the underlying network topology connecting the agents in (2). To solidify this concept, we use the term *mixing matrix* from [24] and state the following assumptions on $\mathbf{W}$.

Assumption 1. *Let $\mathcal{G}_{comm} = (\mathcal{V}_{comm}, \mathcal{E}_{comm})$ represent the network topology of the agents in (2). Then $\mathbf{W}$, the mixing matrix for $\mathcal{G}_{comm}$, has the following properties:*

i. ***(decentralized)*** *if $(i, j) \in \mathcal{E}_{comm}$, then $w_{ij} > 0$, otherwise $w_{ij} = 0$,*

ii. ***(symmetric)*** $\boldsymbol{W} = \boldsymbol{W}^\top$,
iii. ***(null space)*** $\text{null}(\boldsymbol{I} - \boldsymbol{W}) = \text{span}(\boldsymbol{e})$, *where* $\boldsymbol{e}$ *is the vector of all ones, and*
iv. ***(spectral)*** *the eigenvalues of* $\boldsymbol{W}$ *lie in the range* $(-1, 1]$.

Now that the mixing matrix has been defined, we state a matrix version of the DGD algorithm at the $(k+1)^{th}$–iterate:

$$\boldsymbol{\Theta}^{k+1} = \mathbf{W}\boldsymbol{\Theta}^k - \alpha_k \nabla F(\boldsymbol{\Theta}^k) \tag{8}$$

where $\boldsymbol{\Theta}^k \triangleq \begin{bmatrix} (\boldsymbol{\theta}_1^k)^\top \\ \vdots \\ (\boldsymbol{\theta}_N^k)^\top \end{bmatrix} \in \mathbb{R}^{N\times p}$ is a stacking of each local agent's variables into a matrix, $\nabla F(\boldsymbol{\Theta}^k) \triangleq \begin{bmatrix} \nabla f_1(\boldsymbol{\theta}_1^k)^\top \\ \vdots \\ \nabla f_N(\boldsymbol{\theta}_N^k)^\top \end{bmatrix} \in \mathbb{R}^{N\times p}$ is a stacking of each local agent's local gradients *evaluated at their local variable*, and $\alpha_k > 0$ is the step-size. Algorithm 1 shows the DGD updates in (8) from an agent perspective.

Algorithm 1: Local agent view of DGD

Input: Initial local variable: $\boldsymbol{\theta}_i^1 \in \mathbb{R}^p$ for all $i = 1, \dots, N$, maximum iterations: $K > 1$, step-size schedule: $\{\alpha_k\}_{k=1}^K$, neighbor weights: $\{w_{ij}\}_{j\in\mathcal{N}(i)}$ for all $i = 1, \dots, N$, local datasets: $\{(\mathbf{x}_d, y_d)\}_{d\in\mathcal{D}_i}$ for all $i = 1, \dots, N$, training data indices: $\mathcal{T}_i$
Output: Final variable: $\boldsymbol{\theta}_i^K$ for all $i = 1, \dots, N$

for $k = 1, \dots, K$ **do**
  for $i = 1, \dots, N$ *in parallel* **do**
    Compute a local gradient, using the training data $\nabla f_i\left(\boldsymbol{\theta}_i^k\right) \leftarrow \sum_{d\in\mathcal{T}_i} \nabla \ell(\mathbf{x}_d, y_d; \boldsymbol{\theta}_i^k)$
    Communicate with local neighbors to have, $\boldsymbol{\theta}_i^{k+\frac{1}{2}} \leftarrow \sum_{j\in\mathcal{N}(i)} w_{ij}\boldsymbol{\theta}_j^k$
    Update the local parameter, $\boldsymbol{\theta}_i^{k+1} \leftarrow \boldsymbol{\theta}_i^{k+\frac{1}{2}} - \alpha_k \nabla f_i\left(\boldsymbol{\theta}_i^k\right)$

As stated in the introduction, other algorithms use far more powerful techniques to solve (2), but we restrict the focus of this work to using only DGD; this is motivated by two factors:

- DGD requires minimum computation and communication overhead *during each algorithm update*, and
- DGD has proven convergence results in the non-convex setting of (2), where each f_i is a non-convex function.

First analyzed in the convex setting in [23], the authors show that with a constant step-size, i.e. $\alpha_k = \alpha$ for all $k = 1, \dots, K$, then a *near* optimal point can be reached with error $O\left(\frac{\alpha}{\rho}\right)$ where

$$\rho \triangleq 1 - \lambda_+(\mathbf{W}) = 1 - \max\{|\lambda_2(\mathbf{W})|, |\lambda_N(\mathbf{W})|\} \tag{9}$$

measures the connectivity of the graph. If $\mathbf{W} = \frac{1}{N}\mathbf{e}\mathbf{e}^\top$, i.e. the communication graph is fully connected, then $\rho = 1$. The quantity (9) is crucial in the analysis of decentralized methods, see, e.g. [11, 20, 21, 23, 24]. Clearly then, the results of DGD in the convex setting are not promising as $\frac{\alpha}{\rho} \neq 0$. To remedy this issue, diminishing step-sizes are often used in practice to ensure that the optimal solutions can be reached.

The authors in [24] extended DGD's convergence results to the non-convex setting; for sake of completeness, we include statements of their assumptions and their final convergence theorem.

Assumption 2. *The objective functions in (2) satisfy:*

i. f_i *is Lipschitz differentiable with constant* $L_{f_i} > 0$, *i.e.* ∇f_i *satisfies*

$$\|\nabla f_i(\boldsymbol{\theta}_1) - \nabla f_i(\boldsymbol{\theta}_2)\| \leq L_{f_i}\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|, \tag{10}$$

ii. *the gradients of* f_i *are universally bounded by a constant* $B \in (0, +\infty)$, *i.e.* $\|\nabla f_i\| \leq B < +\infty$, *and*
iii. f_i *is coercive, i.e. if* $\|\boldsymbol{\theta}\| \to +\infty$, *then* $f_i(\boldsymbol{\theta}) \to +\infty$.

Given Assumptions 1 and 2, we are now ready to state the theoretical results of DGD on (2).

Theorem 3.1. *(See [24], Theorem 1) Let* $\{\boldsymbol{\Theta}^k\}_{k=1}^K$ *be the sequence generated by DGD in (8) with fixed step-size* $\alpha_k = \alpha < \frac{2-\rho}{\max_i L_{f_i}}$, *and let Assumptions 1 and 2 hold. Then the convergence rate of the sequence*

$$\left\{\frac{1}{K}\sum_{k=1}^K \|\frac{1}{N}\sum_{i=1}^N f_i(\boldsymbol{\theta}_i^k)\|^2\right\}_{k=1}^K = O\left(\frac{1}{K}\right). \tag{11}$$

Additionally, defining $\bar{\boldsymbol{\theta}}^k \triangleq \frac{1}{N}\sum_{i=1}^N \boldsymbol{\theta}_i^k$ *to be the average parameter at the* k^{th} *iterate, then the consensus violation at every iterate satisfies*

$$\|\boldsymbol{\theta}_i^k - \bar{\boldsymbol{\theta}}^k\| \leq \frac{\alpha B}{\rho} \tag{12}$$

for all $i = 1, \dots, N$.

Notice that (11) implies that there exists a $k_0 \in \{1, \dots, K\}$ such that

$$\|\frac{1}{N}\sum_{i=1}^N f_i(\boldsymbol{\theta}_i^{k_0})\|^2 = O\left(\frac{1}{K}\right), \tag{13}$$

thus stationarity for the (2) can be found to ϵ accuracy in $O\left(\frac{1}{\epsilon}\right)$ DGD iterations. While Theorem 3.1 does apply to the problem of non-convex (2), it does not exactly apply to the GCN node classification problem. It is worth noting that the ReLU activation function in 6 is not actually differentiable at the origin; thus the neural network structure considered in (6) does not actually satisfy any of the points in Assumption 2. Nevertheless, DGD exhibits good numerical performance on the node classification problem. The first and third points in Assumption 2 are standard in the literature for non-convex problems, while Assumption 2 *ii.* is strong and greatly limits the scope of theoretically applicable problems for which DGD can be used. It remains open whether or not DGD (and in general, decentralized methods) converge on non-convex, non-Lipschitz differentiable functions.

3.0.1 On the mixing matrix. The final challenge to discuss with DGD is the creation of $\mathbf{W}$. Typically, for problem (2), it is not assumed that the agents have a choice for how to construct which elements $\mathbf{W}$ are non-zero [20, 23]. Following this logic, we note that the use of halo nodes (see Section 2.1 for details) provides a natural way to construct $\mathbf{W}$ for the decentralized node classification problem. Since a halo node provides a link between one agent's data and another agent's data, we take the construction of $\mathbf{W}$ to be:

> $w_{ij} > 0$ if agents i and j are connected via an edge corresponding to a k-hop halo node and $w_{ij} = 0$ if no edge from a k-hop halo node exists between agents i and j.

As an example, consider the graph in Figure 2 where single-hop halo nodes are included. If we let agent 1 be the purple data, agent 2 be the blue data, agent 3 be the orange data, and agent 4 be the green data, then the mixing matrix associated with the partitioned data will be,

$$\mathbf{W}_{\text{example}} = \begin{bmatrix} w_{11} & w_{12} & 0 & 0 \\ w_{21} & w_{22} & w_{23} & 0 \\ 0 & w_{32} & w_{33} & w_{34} \\ 0 & 0 & w_{43} & w_{44} \end{bmatrix}$$

where the actual non-zero elements can be chosen such that Assumption 1 holds. Some examples for how to choose the individual weights (i.e. w_{ij}) once the data has been partitioned, are given by (see [20]):

- (**Laplacian constant edge weights**)

$$w_{ij} = \begin{cases} 1 - \frac{d_{ii}}{\max_i\{d_{ii}\}} & j = i \\ \frac{1}{\max_i\{d_{ii}\}} & (i,j) \in \mathcal{E}_{\text{comm}} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

  where $d_{ii} = \deg(i) = |\mathcal{N}(i)|$ is the degree of agent i, and
- (**Metropolis constant edge weights**)

$$w_{ij} = \begin{cases} 1 - \sum_{n=1}^{N} w_{in} & j = i \\ \frac{1}{\max\{\deg(i),\deg(j)\}+\epsilon} & (i,j) \in \mathcal{E}_{\text{comm}} \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

  where $\epsilon > 0$ is some constant, typically chosen to be 1.

For us, we choose the weights to be a variation of (14). Namely, if $\bar{\mathbf{W}}$ gives the weights from (14), then we choose $\mathbf{W} = \left(\mathbf{I} + \bar{\mathbf{W}}\right)/2$ where $\mathbf{I}$ is the $N \times N$ identity matrix. Notice that if $\bar{\mathbf{W}}$ satisfies Assumption 1, so does $\mathbf{W} = \left(\mathbf{I} + \bar{\mathbf{W}}\right)/2$. Thus, $\mathbf{W}$ is simply a re-weighting that enforces that mixing matrix is *positive semi-definite*. This is not theoretically required for DGD, but we choose this formulation here as it provides good numerical performance. We are now in position to show discuss the performance of DGD on the node classification problem.

4 NUMERICAL EXPERIMENTS

We test Algorithm 1 on the Cora citation graph dataset, which is a standard node classification dataset [9, 17]. The Cora dataset consists of 2708 scientific publications and 7 possible classes. There are 5429 edges in the dataset and each $\mathbf{x}_j$ is a binary vector in 1433-dimensional space representing a dictionary of words, where $(\mathbf{x}_j)_d$, i.e. the d^{th} component of $\mathbf{x}_j$, is 1 if the word appears in the publication and 0 if it does not. Similar to [9], we use 140 training data points (i.e. $|\mathcal{T}| = 140$), so that each class has 20 representative vectors scattered throughout the graph. Our experiments are focused on how training time and testing accuracy is affected by the use of graph partitioning as described in Section 2.1. As discussed in the previous section, the construction of $\mathbf{W}$ used in (8) is based upon the underlying structure of the Cora dataset and where halo nodes exist after partitioning. We run experiments for $N = 1, 4, 8, 16, 32$ in (2). Figure 4 shows the non-zero entries of $\mathbf{W}$ for each chosen N (for $N = 1$, we choose $\mathbf{W} = w_{11} = 1$). A black square in the $(i,j)^{th}$ location indicates that agents i and j share a 2-hop halo node edge, as 2-hop information is required for (6) (see Section 2.1 for a more detailed explanation). An example of how 2-hop halo nodes affect the size of the local agent's dataset is given in Figures 5 and 6; this is for the case $N = 32$ and the gray dots represent nodes in the local agent's dataset.

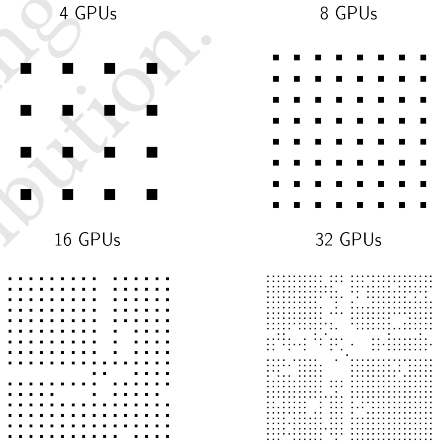

Figure 4: Sparsity patterns for the mixing matrices associated with varying number of GPUs.

The Cora dataset is densely structured as a non-fully connected $\mathbf{W}$ does not appear until there 16 agents (see Figure 4). The addition of more agents will result in an even more sparsely connected $\mathbf{W}$ and future work will explore the benefits of experimenting with more agents. Before discussing the performance of DGD on the node classification problem, experimental details are given.

4.1 mpi4py

The novelty of this work is in the construction of a non-blocking send and receive style function implemented in PyThon using mpi4py, a PyThon wrapping of MPI [3], that is compatible with PyTorch. As stated in Section 2.1, PyTorch tensors have a `.numpy()` method that allows tensors to be converted to Numpy arrays. While mpi4py is compatible with general PyThon objects, it has the fastest performance on Numpy arrays [3]. PyTorch tensors are most easily extracted from a model in a *list* format, thus each element within the

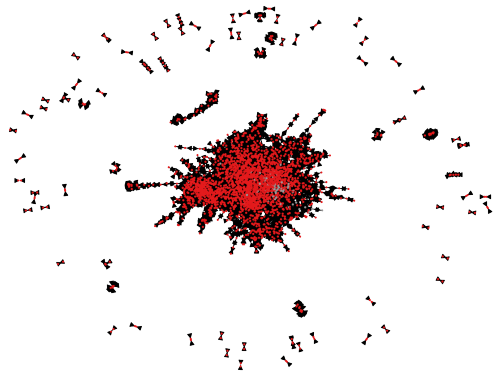

Figure 5: Example of one local agent's data in a 32 agent structure BEFORE inclusion of 2-hop halo nodes. Gray dots are the nodes on this agent.

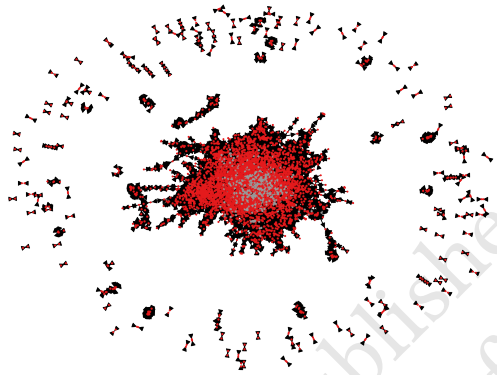

Figure 6: Example of one local agent's data in a 32 agent structure AFTER 2-hop inclusion of halo nodes. Gray dots are the nodes on this agent.

list must be converted to a Numpy array before being sent to neighboring agents. Algorithm 2 displays the details of the non-blocking send and receive function utilized in this work.

Looping over all of the parameters in the neural network, i.e. $\boldsymbol{\theta}$, Algorithm 2 does four things at every invocation:

(1) convert the current parameter to a numpy array and allocate enough space in memory to receive an array from every neighbor (lines 2 – 6),
(2) perform a non-blocking send to every neighbor of agent i (lines 7 and 8),
(3) perform a non-blocking receive from every neighbor of agent i, filling the appropriate location in the allocated space in memory (lines 9 and 10), and
(4) wait until all information has been sent and received before performing a neighbor aggregation (lines 11 – 14).

Algorithm 2: mpi4py neighbor communication algorithm (local agent view)

Input: List of tensors to send: $\boldsymbol{\theta}_i$, neighbors: $\mathcal{N}(i)$, neighbor weights: $\{w_{ij}\}_{j\in\mathcal{N}(i)}$
Output: Neighbor averaged parameter: $\boldsymbol{\theta}_i^+$

1 **for** $\boldsymbol{\phi} \in |\boldsymbol{\theta}_i|$ **do**
2 Allocate an empty numpy array to receive neighbor parameters, $\boldsymbol{\phi}_{\text{receive}}(i)$, of size
3 $$\left(|\mathcal{N}(i)| \cdot \boldsymbol{\phi}.\text{shape}[0], \boldsymbol{\phi}.\text{shape}[1]\right)$$
4 where $\boldsymbol{\phi}$.shape[t] indicates the length of the t^{th}-dimension of parameter $\boldsymbol{\phi}$
5 Convert the parameter $\boldsymbol{\phi}$ to a numpy array to be sent
6 $$\boldsymbol{\phi}_{\text{send}}(i) \leftarrow \boldsymbol{\phi}$$
7 **for** $j \in \mathcal{N}(i)$ **do**
8 Use mpi4py's `.Isend()` to send $\boldsymbol{\phi}_{\text{send}}(i)$ to agent j
9 **for** $j \in \mathcal{N}(i)$ **do**
10 Use mpi4py's `.Irecv()` to receive $\boldsymbol{\phi}_{\text{send}}(j)$ in rows $j \cdot \boldsymbol{\phi}.\text{shape}[0]$ to $(j+1) \cdot \boldsymbol{\phi}.\text{shape}[0]$ of $\boldsymbol{\phi}_{\text{receive}}(i)$
11 Perform an `MPI.Request.waitall()` on the return value of the `Isend()` and `Irecv()` methods
12 Aggregate the information to update the parameter:
13 $$\boldsymbol{\theta}_i^+(\boldsymbol{\phi}) \leftarrow w_{ii}\boldsymbol{\phi} + \sum_{j\in\mathcal{N}(j)} w_{ij}\boldsymbol{\phi}_{\text{receive}}(i)$$
14 where it is assumed that $\boldsymbol{\phi}_{\text{receive}}(i)$ is re-shaped to have w_{ij} applied to the appropriate rows and also converted back to a tensor

It is worth noting that the size of the neural network in (6) is small (there are only $1433 \times h_2 + h_2 \times 7 = 1440 \cdot h_2$ parameters where h_2 is the dimension of the hidden layer in the network) and as such can be easily trained on a CPU in 4 seconds [9]. So not only does this work display the effect of using DGD to solve the node classification problem, it also quantifies how using GPUs affect the corresponding training time and the classification accuracy.

4.2 Performance

All experiments are ran on the AiMOS supercomputer at Rensselaer Polytechnic Institute. We utilize the NPL cluster which has 40 computing nodes. Each computing node has 8 NVIDIA Tesla V100 GPUs and dual 100 gigabyte EDR InifiniBand cables facilitating the connection between the GPUs.

In order to be consistent with [9], we run Algorithm 1 for $K = 350$ iterations for all experiments. As mentioned previously, we vary $N \in \{1, 4, 8, 16, 32\}$ to study the effects that partitioning graph data has on training and testing metrics. For the GCN architecture (6), we fix the dimension of the hidden layer to be 64 so that $\boldsymbol{\theta}_i^{(1)} \in \mathbb{R}^{1443\times 64}$ and $\boldsymbol{\theta}_i^{(2)} \in \mathbb{R}^{64\times 7}$ for all $i = 1, \ldots, N$, making the model have a total

of 92,160 learnable parameters. This model is *larger* than the GCN used in [9], but we find it has good performance and is inspired by the choice of hidden layer size in [17]. We choose a constant step-size in (8) so that $\alpha_k = \alpha$ for all $k = 1, \dots, K$. This parameter is optimized for each N.

As seen in Figure 4, as N increases, so does the size of $\mathcal{N}(i)$ for all $i = 1, \dots, N$. Figure 7 shows how the time spent running Algorithm 1 is divided between *communication*, i.e. time spent in Algorithm 2, and *computation*, i.e. time spent computing gradients and updating the local parameters. Since $\mathbf{W}$ is not exceptionally sparse for any of our experiments, it is clear that Algorithm 2 dominates the total time taken to train the GCN. In the case with $N = 32$, DGD actually doubled the training time from that reported in [9].

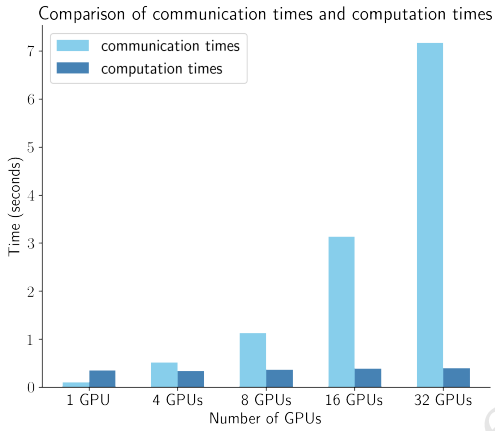

Figure 7: Time difference between computation time (lines 3 and 5 of Algorithm 1) and communication time (line 4 of Algorithm 1).

The benchmark testing accuracy for the Cora dataset is 81.5% on 1000 randomly selected data points. To show the efficacy of DGD, Figure 8 reports the testing accuracy for each N. Since we have fixed the iteration number to 350, the 16 GPU and 32 GPU cases are not able to reach the benchmark accuracy in the alloted time. We make note however, that DGD has *significant* improvement over the algorithm proposed in [17]. While Algorithm 1 does make use of GPUs, the x-axis scale for both this paper and [17] is in terms of *iterations*, with the scale in [17] taking roughly 10 times more iterations to reach a testing accuracy of above 50% for all N experimented on. This suggest that the method of partitioning proposed in Section 2.1 is not only logical, but also efficient in practice. For the fully connected cases of $N = 1, 4, 8$, DGD is competitive with the centralized method from [9], as it is able to reach the benchmark accuracy before 300 iterations. For sparser communication topologies, DGD needs more iterations to reach the desired benchmark accuracy. For sake of completeness, we also include the testing loss for varying N in Figure 9.

4.2.1 On strong scaling. The final comparison of importance to us is demonstrating the effect that *strong scaling* has on the running time of Algorithm 1. As stated in [9], the training time for a GCN

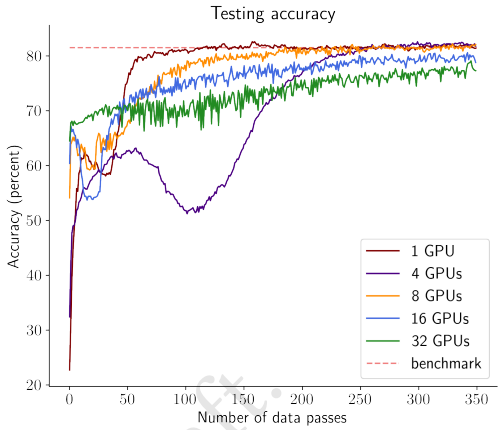

Figure 8: Testing accuracy on 1000 data points partitioned among N agents (each GPU is an agent). Benchmark accuracy is from [9].

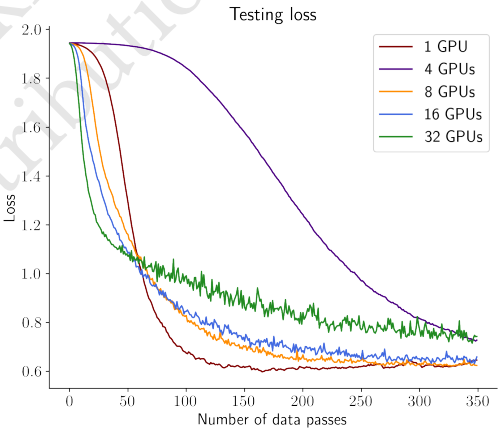

Figure 9: Testing loss on 1000 data points partitioned among N agents (each GPU is an agent).

of the form (6) with 46,080 parameters (half of the parameters used in this work) on a CPU takes 4 seconds to complete 350 iterations (these are Adam [8] updates). Figures 10 and 11 display the total training time to reach the benchmark accuracy, as well as the resulting speed up over the CPU case from [9]. Here, we say that DGD has "reached benchmark accuracy" at iteration k if iterates $k + 1, \dots, k + 10$ all have accuracy greater than or equal to the benchmark. Figure 10 shows that performing the same training with a GPU decreases the training time to less than 1 second, resulting in $\approx 20\times$ increase in training speed; see Figure 11. As the number of GPUs increases and hence the size of $\mathbf{W}$ increases, the training time (speedup factor) increases (decreases). In the centralized setting, there is no waiting for communication between agents, hence Algorithm 2 is not necessary so training time is saved. We

conjecture that if $N >> 100$, then the sparsity structure of $\mathbf{W}$ would be conducive to illustrating an improvement with the use of DGD.

4.2.2 On weak scaling. A *weak scaling* study is not performed during this work as finding appropriately sized datasets for intermediate communication graph sizes (e.g. $N = 16, 32$) is challenging. It is worth noting that the other benchmark datasets in [9] contain either $O(1,000)$ nodes (e.g. the Cora dataset utilized in this work), or $O(10,000)$ nodes (e.g. the Pubmed dataset [9]), which would require many GPUs (e.g. $N >> 100$) to see the effect that decentralized training has on datasets of this size. Limited by the number of GPUs available to us, we remark that a weak scaling study could be more easily performed with access to more GPUs.

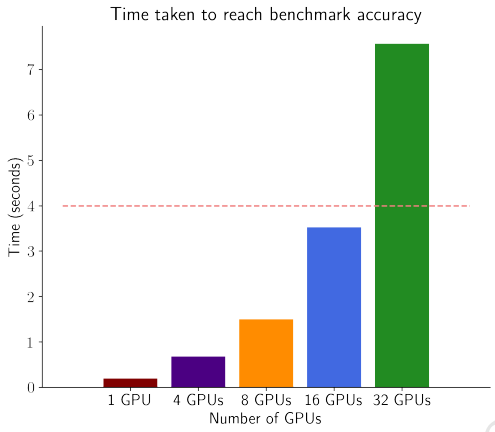

Figure 10: Time required to reach benchmark testing accuracy of 81.5%. 16 GPUs and 32 GPUs did not actually meet the benchmark accuracy in the allowed number of data passes.

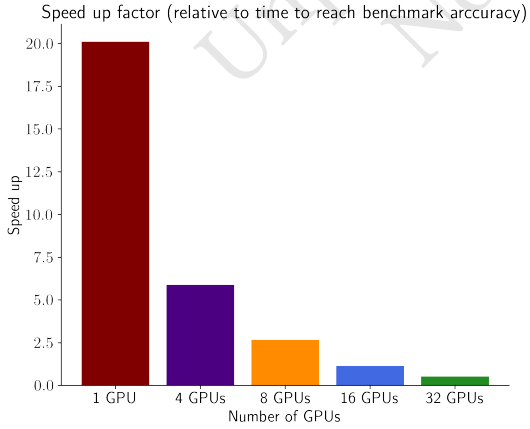

Figure 11: Speedup performance relative to benchmark time of 4 seconds (see [9]) to reach 81.5% testing accuracy.

5 CONCLUSION

This work adapts a commonly used decentralized optimization method (DGD) to the problem of training a GCN for solving node classification. A literature review of both decentralized optimization methods and graph neural network architectures is presented in Section 1.1, followed by a detailed explanation of both the GCN architecture and the DGD method (Sections 2 and 3, respectively). The process of distributing graph data among a set of N agents is accomplished through the use of PyMetis, and a novel communication algorithm using mpi4py is given in Algorithm 2. Numerical performance of DGD on the node classification problem is shown in Section 4.2, where the speed improvement and the effect on testing accuracy is discussed. The work presented here suggests that partitioning a graph and using halo nodes provides an advantage over other methods that do not do so, see [17] Figure 3 to compare to the results from this work. Future work includes testing Algorithms 1 and 2 in a setting where $N >> 1$, e.g. $N = 1000$, and applying more robust algorithms such as Prox-PDA [6] to solve the node classification problem. Additionally, developing and analyzing theoretically robust decentralized methods that allow non-differentiable local loss functions is another target of future work.

REFERENCES

[1] Yossi Arjevani, Joan Bruna, Bugra Can, Mert Gurbuzbalaban, Stefanie Jegelka, and Hongzhou Lin. Ideal: Inexact decentralized accelerated augmented lagrangian method. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 20648–20659. Curran Associates, Inc., 2020.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Inc., USA, 1989.

[3] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011. New Computational Methods and Software Tools.

[4] M. Dawodi, J. A. Baktash, T. Wada, N. Alam, and M. Z. Joya. Dari speech classification using deep convolutional neural network. In *2020 IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS)*, pages 1–4, 2020.

[5] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[6] Mingyi Hong, Davood Hajinezhad, and Ming-Min Zhao. Prox-PDA: The proximal primal-dual algorithm for fast distributed nonconvex optimization and learning over networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1529–1538, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[7] George Karypis and Vipin Kumar. MeTis: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0, 2009.

[8] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR (Poster)*, 2015.

[9] Thomas Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*, 5, 2017.

[10] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[11] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 5330–5340. Curran Associates, Inc., 2017.

[12] Xiangru Lian, Wei Zhang, Ce Zhang, and Ji Liu. Asynchronous decentralized parallel stochastic gradient descent. *Proceedings of the 35th International Conference on Machine Learning*, 80:3043–3052, 2018.

[13] P. D. Lorenzo and G. Scutari. Next: In-network nonconvex optimization. *IEEE Transactions on Signal and Information Processing over Networks*, 2(2):120–136, 2016.

[14] Brendan McMahan and Daniel Rampage. Federated learning: Collaborative machine learning without centralized training data. *Google AI Blog*, 2017.

[15] Angelia Nedic and Asuman Ozdaglar. Distributed subgradient methods for multi-agent optimization. *IEEE Transactions on Automatic Control*, 54:48 – 61, 2009.
[16] S. Pu, W. Shi, J. Xu, and A. Nedić. Push–pull gradient methods for distributed optimization in networks. *IEEE Transactions on Automatic Control*, 66(1):1–16, 2021.
[17] Simone Scardapane, Indro Spinelli, and Paolo Di Lorenzo. Distributed training of graph convolutional networks. *IEEE Transactions on Signal and Information Processing over Networks*, 7:87–100, 2021.
[18] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.
[19] W. Shi, Q. Ling, K. Yuan, G. Wu, and W. Yin. On the linear convergence of the admm in decentralized consensus optimization. *IEEE Transactions on Signal Processing*, 62(7):1750–1761, 2014.
[20] Wei Shi, Qing Ling, Gang Wu, and Wotao Yin. Extra: An exact first-order algorithm for decentralized consensus optimization. *SIAM Journal on Optimization*, 25:944 – 966, 2015.
[21] Haoran Sun, Songtao Lu, and Mingyi Hong. Improving the sample and communication complexity for decentralized non-convex optimization: Joint gradient estimation and tracking. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9217–9228, Virtual, 13–18 Jul 2020. PMLR.
[22] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019.
[23] Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26:1835 – 1854, 2016.
[24] Jinshan Zeng and Wotao Yin. On nonconvex decentralized gradient descent. *IEEE Transactions on Signal Processing*, 66:2834 – 2848, 2018.
[25] Jiaqi Zhang and Keyou You. Decentralized stochastic gradient tracking for non-convex empirical risk minimization. *arXiv:1909.02712*, 2020.
[26] Jiaqi Zhang and Keyou You. Fully asynchronous distributed optimization with linear convergence in directed networks, 2021.
[27] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. Distdgl: Distributed graph neural network training for billion-scale graphs, 2020.