DGCL: An Efficient Communication Library for Distributed GNN Training

Zhenkun Cai
The Chinese University of Hong Kong
zkcai@cse.cuhk.edu.hk

Xiao Yan*
Southern University of Science and Technology
yanx@sustech.edu.cn

Yidi Wu
The Chinese University of Hong Kong
ydwu@cse.cuhk.edu.hk

Kaihao Ma
The Chinese University of Hong Kong
khma@cse.cuhk.edu.hk

James Cheng
The Chinese University of Hong Kong
jcheng@cse.cuhk.edu.hk

Fan Yu
Huawei Technologies Co. Ltd
fan.yu@huawei.com

Abstract

Graph neural networks (GNNs) have gained increasing popularity in many areas such as e-commerce, social networks and bio-informatics. Distributed GNN training is essential for handling large graphs and reducing the execution time. However, for distributed GNN training, a peer-to-peer communication strategy suffers from high communication overheads. Also, different GPUs require different remote vertex embeddings, which leads to an irregular communication pattern and renders existing communication planning solutions unsuitable. We propose the **d**istributed **g**raph **c**ommunication **l**ibrary (DGCL) for efficient GNN training on multiple GPUs. At the heart of DGCL is a communication planning algorithm tailored for GNN training, which jointly considers fully utilizing fast links, fusing communication, avoiding contention and balancing loads on different links. DGCL can be easily adopted to extend existing single-GPU GNN systems to distributed training. We conducted extensive experiments on different datasets and network configurations to compare DGCL with alternative communication schemes. In our experiments, DGCL reduces the communication time of the peer-to-peer communication by 77.5% on average and the training time for an epoch by up to 47%.

Keywords Graph Neural Networks, Distributed and Parallel Training, Network Communication

*Corresponding author.

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456233

ACM Reference Format:
Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3447786.3456233

1 Introduction

Graph neural networks (GNNs) are a special kind of deep neural network, in which each vertex aggregates the embeddings of its neighbors in the graph to compute its own embedding. Many GNN models have been proposed, e.g., GCN [17], GAT [33], CommNet [32], GraphSAGE [8], GIN [39] and GGNN [20]. These GNN models achieve excellent performance for graph related tasks such as node classification, link prediction and graph clustering. To support the efficient training of GNN models, a number of GNN systems such as DGL [35], PyG [7], NeuGraph [21], RoC [13] and AliGraph [40], have been developed for CPU or GPU. As real graphs can be very large, e.g., containing millions of vertices and billions of edges, it is essential to conduct distributed GNN training using many GPUs for efficiency and scalability. However, most existing GNN systems are designed for a single worker (e.g., DGL and PyG) or a small number of workers on the same machine (e.g., NeuGraph).

To conduct distributed GNN training, a graph is usually partitioned [4, 14, 15, 30] to assign its vertices to multiple workers, and the workers compute the vertex embedding in parallel. As vertices need to access the embeddings of their neighbors that reside on other workers, it is necessary to conduct embedding passing during training. However, using direct peer-to-peer communication (i.e., each worker fetches the required embeddings directly from other workers) [12] for embedding passing results in high communication overheads. The communication time can easily take up over 50% of the total training time and even exceeds 90% in some cases according to our measurements in §3.

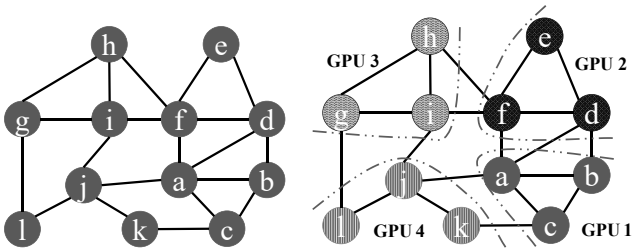
Our profiling and analysis show that peer-to-peer communication has poor efficiency for two main reasons (§3):

(1) It fails to fully utilize fast links. Modern GPU servers (e.g., NVIDIA DGX systems [22]) contain heterogeneous physical connections such as NVLink [25], PCIe, QPI, IB [36] and Ethernet, and the bandwidth of the fastest link can be an order of magnitude of the slowest link. Peer-to-peer communication simply uses the direct links between workers and does not fully exploit the fast links. (2) It does not consider the communications between different worker pairs jointly. With peer-to-peer communication, all workers communicate with each other concurrently for embedding passing. This results in severe network contention and load imbalance given the complex connections among GPUs in a modern GPU cluster. To eliminate embedding passing, we considered an alternative that replicates the remote neighbors of the vertices to the local worker but found that replication results in high memory and computation costs.

To optimize communication efficiency of distributed GNN training, we formulate a *communication planning problem*, which is to find a plan that minimizes the time to pass the required vertex embeddings among the workers. The problem is challenging for two reasons: (1) The communication pattern is *irregular* as different workers require different remote embeddings. This is in contrast to training normal deep neural networks (e.g., VGG [31] and ResNet [9] variants), for which each worker sends/receives the same set of model parameters and communication planning has been solved by NCCL [24]. (2) The physical connections are complex in modern GPU servers and the communications of all worker pairs need to be jointly considered to avoid contention and balance the loads among different links.

To solve the communication planning problem, we first show that the optimal strategy to transmit a vertex embedding from its source worker (i.e., the worker that computes it) to its destination workers (i.e., the workers that use it as input) is a tree that has the source worker as its root and contains all the destination workers. Then, a communication plan is defined as the union of the communication trees of all vertices, which provides the maximum flexibility as each vertex is allowed to choose its own communication strategy. We also build a cost model to predict the execution time of a communication plan, which divides the communications into stages and considers contention, load balancing and parallelization. Finally, we propose a *shortest path spanning tree* (*SPST*) algorithm to minimize the cost model in a greedy manner. The vertices are considered sequentially and the communication tree for each vertex is solved by minimizing the cost blow-up considering the processed vertices.

Based on the SPST algorithm, we develop a package called the *distributed graph communication library* (DGCL). DGCL handles tasks related to distributed GNN training including graph partitioning, communication planning, and actual communication execution. Existing GNN systems can easily invoke DGCL with user-friendly APIs for efficient distributed communication. The GNN system only needs to work in a single-worker mode and does not need to know the details about distributed execution. In addition to the SPST algorithm, we introduce several system designs in DGCL to improve communication efficiency, including decentralized communication coordination, automatic communication method selection and non-atomic aggregation.



(a) Graph **(b)** Partition to 4 GPUs

Figure 1. An example graph and its partitioning

To evaluate the performance of DGCL, we conducted extensive experiments on four graphs and three GNN models under different network configurations. We compared with three alternative communication schemes, i.e., *peer-to-peer communication*, *swap* (which uses main memory for embedding exchange among GPUs as in NeuGraph [21]) and *replication* (which replicates cross-partition vertices to eliminate communication as in Medusa [43]). The results show that DGCL consistently achieves shorter communication time and hence shorter per-epoch time than the baselines for distributed GNN training. Compared with peer-to-peer communication, DGCL reduces the communication time by at most 85.8% and 77.5% on average. The per-epoch time of peer-to-peer is reduced by up to 47% in some cases. We also found that replication has poor performance in processing dense graphs and runs out of memory for large graphs. Swap often has the worst performance as it needs to dump all vertex embeddings to main memory.

Paper outline. We give the background on GNN training in §2. We analyze existing strategies for distributed GNN training in §3 to show their limitations and hence motivate our work. We present the architecture and the API of DGCL in §4. We discuss the details of communication planning and the SPST algorithm in §5. We discuss the system designs and some implementation details in §6. We report the performance evaluation results in §7. We discuss related work in §8 and conclude our work in §9.

2 Background on GNN Training

Given a graph $G = (V, E)$, GNN models (e.g., GCN [17], CommNet [32] and GIN [39]) learn an embedding for each vertex $v \in V$ by stacking multiple graph propagation layers.

In each layer, GNN models generally follow an *aggregate-update* computation pattern:

$$\begin{aligned} a_v^{(k)} &= \text{AGGREGATE}^{(k)}(\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}) \\ h_v^{(k)} &= \text{UPDATE}^{(k)}(a_v^{(k)}, h_v^{(k-1)}) \end{aligned}, \quad (1)$$

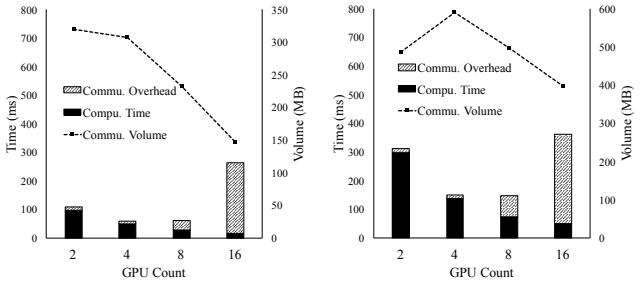
where $h_v^{(k)}$ is the embedding of vertex v in the k-th layer and $h_v^{(0)}$ is the input features of vertex v. The $\text{AGGREGATE}^{(k)}$ function collects the embeddings of v's neighbors in $(k-1)$-th layer, i.e., $\{h_u^{(k-1)} | u \in \mathcal{N}(v)\}$, to compute $a_v^{(k)}$. The $\text{UPDATE}^{(k)}$ function uses the neighborhood aggregation result $a_v^{(k)}$ and v's embedding in the $(k-1)$-th layer, i.e., $h_v^{(k-1)}$, to calculate v's embedding in the k-th layer. Both the AGGREGATE and UPDATE functions can be neural networks, which are updated during training. Although $K=2$ layers are most popular, recent works suggest that 3 or more layers provides better performance for some tasks [10, 13].

GNN training is conducted by alternating between forward passes and backward passes. In a forward pass, each vertex aggregates the embeddings of its neighbors to compute its final output (i.e., the K-th layer embedding). In a backward pass, each vertex sends gradients to its neighbors to update the AGGREGATE and UPDATE functions. We complete an *epoch* of training when all vertices have gone through a forward and backward pass. Beside the *full graph training* method, some works use *sampling* [8], which only computes the final output for some vertices in a pass. As sampling has potential accuracy loss [13], we consider full graph training in this paper. For a K-layer GNN, the computation of a vertex needs to access the embedding of its K-hop neighbors. Consider a 2-layer GNN and take vertex a in Figure 1a for example. Computing $h_a^{(1)}$ requires the 0-th layer embeddings of a's direct neighbors $\mathcal{N}(a)=\{b, c, d, f, j\}$, and $h_a^{(2)}$ requires the 1st layer embeddings of the vertices in $\mathcal{N}(a)$, which in turn depends on the 0-th layer embeddings of their direct neighbors. As a result, training a involves all the vertices in the example graph except vertex g, which is 3 hops away from a.

3 An Analysis on Strategies for Distributed GNN Training

In this section, we give an analysis on various strategies for distributed GNN training to show the limitations of existing methods and the challenges of distributed GNN training.

Graph partitioning without replication. A straightforward scheme for distributed GNN training is *graph partitioning* as illustrated in Figure 1b. The graph is partitioned into non-overlapping partitions (i.e., without vertex replication on different workers), and each GPU conducts computation for a partition. Execution follows a *transfer-compute* schedule for each layer, in which the embeddings of the required non-local vertices are fetched from remote GPUs before conducting computation. The cost for *embedding passing* can be high as the vertex embeddings are high-dimensional vectors and many vertex embeddings need to be transferred.



(a) Web-Google **(b)** Reddit

Figure 2. The computation time and communication overhead for training a 2-layer GCN with peer-to-peer communication (the average communication volume of each GPU during an epoch is plotted as dashed lines)

Table 1. The speed (GBps) of common communication links (NV2 and NV1 mean 2 and 1 NVLinks between two GPUs)

Type	NV2	NV1	PCIe	QPI	IB	Ethernet
Speed	48.35	24.22	11.13	9.56	6.37	3.12

We plot the computation time, communication overhead and average communication volume of each GPU for training a 2-layer GCN in Figure 2. [1] The graph is partitioned using the METIS library [14] to minimize the number of cross-partition edges for communication reduction, and the GPUs transfer the vertex embeddings to each other via *peer-to-peer* communication. Figure 2 shows that the communication time grows rapidly with the number of GPUs, taking up more than 50% of the per-epoch time for both graphs with 8 GPUs. With 16 GPUs on two machines, the communication time takes up more than 90% of the per-epoch time due to slow cross-machine connection. The communication time increases with the number of GPUs even though the per-GPU communication volume decreases, because the aggregated communication volume of all GPUs increases with the number of GPUs and different GPUs may contend for communication, which severely impairs efficiency as we will show shortly.

The communication topology inside one of the machines used in our experiments is plotted in Figure 3, which shows there are a few types of connection between GPUs, e.g., NVlink, PCIe, PCIe-QPI-PCIe. Different types of connections vary significantly in speed according to our measurements

[1] The detailed experiment configurations, e.g., the connections among the GPUs, dataset statistics and mode structures can be found in §7.

Table 2. The time (ms) peer-to-peer communication spends on different links for training a GCN layer with 8 GPUs

	Web-Google	Reddit	Wiki-Talk
NVLink	0.99	1.70	1.39
Others	6.20	18.1	6.13

Table 3. Attainable bandwidth (Gbps) of a GPU when there are different number of GPUs using the QPI link

Number of GPUs	1	2	3
Attainable bandwidth	9.50	5.12	3.34

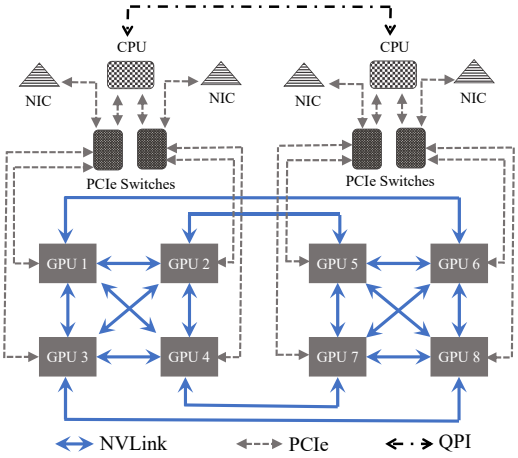
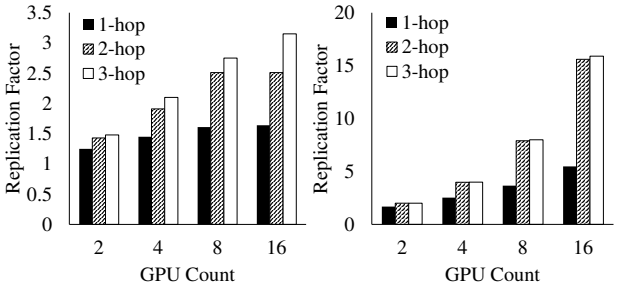


Figure 3. The connections among the GPUs of the NVIDIA DGX-1 system, as an example of the complex communication connections in modern GPU servers

in Table 1. We also profile the time that peer-to-peer communication spends on NVLink and other relatively slow connections when training a GNN layer. As shown in Table 2, the GPU pairs with NVLink connections (e.g., GPU1-GPU2, GPU4-GPU7) have much shorter communication time than those with slow connections (e.g., GUP1-GPU5 via PCIe-QPI-PCIe). As all GPU pairs in Figure 3 can be connected within two hops of NVLink, we can reduce communication time by eliminating data transfer on the slow links and using NVLink-based relay, e.g., assigning GPU2 to forward the communication between GPU1 and GPU5. *Therefore, a key reason behind the long communication time of peer-to-peer is that it simply utilizes the direct links and fails to fully utilize the fast links.*

In Figure 3, the direct communication links from GPU1, GPU3 and GPU4 to GPU5 all go through the QPI, and these GPUs will contend for the QPI bandwidth if they communicate with GPU5 concurrently in peer-to-peer communication. We report the attainable bandwidth for a GPU when there are



(a) Web-Google **(b)** Reddit

Figure 4. The replication factor for training GNNs with different number of layers

different numbers of concurrent GPUs using the QPI link in Table 3. The results show that contention severely degrades communication speed. *Thus, another limitation of peer-to-peer communication is that it does not consider concurrent communications jointly and may result in contention.*

Graph partition with replication. The communication overhead for embedding passing can be completely removed using replication. Beside the vertices in the local partition, each GPU also stores the K-hop neighbors of its local vertices and computes the intermediate embeddings of the replicated vertices when necessary. In this case, each GPU can compute the final outputs for the vertices in its local partition without communication but needs to pay additional storage and computation costs.

We define *replication factor* as the total number of (assigned and replicated) vertices kept by all GPUs divided by the number of vertices in the original graph. Replication factor is an indicator of the computation and storage overheads needed to eliminate embedding passing. Figure 4 plots the replication factor for GNNs with different number of layers (i.e., hops). The results show that the replication factor increases with the number of GPU and GNN layers. For the denser Reddit graph, each GPU almost stores the entire graph for GNNs with more than 1 layers, i.e., the 2-hop neighbors already cover almost the entire graph, and thus the replication factor of 2-hop and that of 3-hop are almost the same in Figure 4b. For the sparser Web-Google graph, the replication factor is also large for a 3-layer GNN (more than 3 when 16 GPUs are used). The reason is that large graphs usually have a small diameter and a vertex can reach many vertices in a few hops. This renders replication inapplicable for deeper GNN models with more layers, which is shown to achieve good performance recently [13]. We will also verify in our experiments in §7 that using replication results in a long per-epoch time due to the repetitive computation of vertex embeddings on different GPUs.

Other options for distributed GNN training. Besides non-overlapping graph partitioning and replication, there are

many other strategies for distributed GNN training that show different interplay among communication, memory and computation overheads. We list some examples: (1) When GPU memory is sufficient, the 0-th layer embeddings of the adjacent vertices can be cached to eliminate embedding passing for the 1st GNN layer. (2) When GPU memory is insufficient (e.g., for large graphs or a small number of GPUs), embeddings need to be offloaded to and reloaded from CPU memory dynamically. (3) When cross-machine bandwidth is low, replication may be adopted across different machines to reduce cross-machine communication, while the GPUs inside the same machine use non-overlapping graph partitioning.

In all these strategies, GNN training still needs to find efficient methods to transfer the required embedding to each GPU, which we call a *communication planning* problem. For example, embedding passing is required for layer-2 and higher layers in (1) above, while the GPUs inside the same machine need to exchange embeddings for (3). For (2), the embedding swap between GPU memory and CPU memory can also be regarded as a special form of communication via PCIe. Although we focus on non-overlapping graph partitioning in our presentation, our communication planning algorithm can be easily generalized to more diverse GNN training strategies.

Properties of GNN communication planing. Communication is *irregular* for distributed DNN training as different GPUs need to send/receive different vertex embeddings from other GPUs according to graph partitioning. In comparison, communication in the data-parallel training of usual neural networks (e.g., those for computer vision) are regular and all GPUs send/receive the same set of model parameters, and existing libraries such as NCCL are well-designed for planning this type of regular communication. Motivated by the limitations of peer-to-peer communication, we should consider all concurrent communications jointly and fully utilize fast links while avoiding contention at the same time. As embedding passing is largely independent from the deep learning part of GNN training (e.g., neural network inference and back propagation), a general communication planning algorithm and execution framework can be reused for different GNN systems.

4 The Architecture and API of DGCL

In this section, we first introduce the architecture and workflow of DGCL, and then present its API and discuss how to connect DGCL with existing GNN systems with the API.

4.1 Architecture and Workflow

The architecture of DGCL is shown in Figure 5. DGCL accepts an input graph and partitions the graph. The number of graph partitions is the same as the number of GPUs as each partition is assigned to one GPU. For graph partitioning, we use the METIS library [14] to minimize the number of

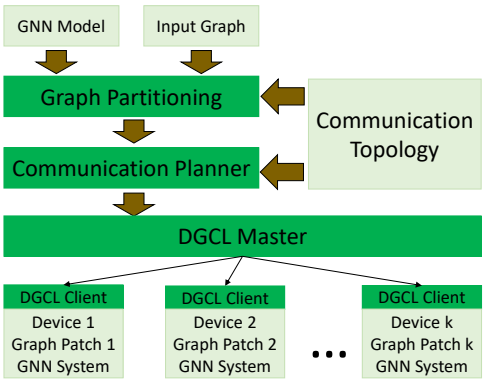


Figure 5. Architecture of DGCL

cross-partition edges for communication reduction and also ensure that each partition has a similar number of vertices for load balancing. There are usually hierarchies in the communication topology, for example, intra-machine connections can be faster than inter-machine connections, and inside the same machine, GPUs on the same NUMA node may have faster connections than those on different NUMA nodes. In these cases, we use hierarchical graph partitioning to prioritize communication reduction on slow links. Currently, we assume that the graph partitioning is non-overlapping, and the GPU memory is sufficient for the graph data and vertex embeddings in each partition.

For each graph partition, the vertices are re-indexed to make distributed training transparent to the underlying GNN system, i.e., the GNN system only sees an input graph instead of a partition. For a GPU d, we use V_d^l to denote its *local vertices* (i.e., vertices in its partition), V_d^r to denote its *remote vertices* (i.e., direct neighbors of its local vertices that reside in other partitions), and E_d to denote its local edges. For example, in the graph partitioning in Figure 1b, we have $V_1^l = \{a, b, c\}$ and $V_1^r = \{d, f, j, k\}$ for GPU 1. For a GPU d, DGCL constructs a graph $G_d(V_d^l \cup V_d^r, E_d)$ and the GNN system can conduct training on G_d in the single-GPU mode when the embeddings of vertices in $V_d^l \cup V_d^r$ are available. The *communication relation* among the GPUs is also constructed according to the graph partitioning result. We record a tuple (d_i, d_j, V_{ij}) for each pair of GPUs, where V_{ij} is the vertex embeddings that GPU d_i needs to send to GPU d_j.

We associate a DGCL client with each GPU, which is responsible for conducting communication for distributed training, e.g., sending/receiving vertex embeddings to/from remote GPUs and forwarding embeddings for other GPUs. A centralized DGCL master monitors the DGCL clients and notifies them to start the communication for a layer or when all communications for a layer finish. The communication planner lies at the heart of DGCL, which takes as input the communication relation among the GPUs and the communication topology, and generates a communication plan for

```
1  import dgl, dgcl
2
3  class distributed_gcn_model(object):
4    def forward(self, embeddings):
5      for l in layers:
6        embeddings = dgcl.graph_allgather(
   embeddings)
7        embeddings = dgl.GraphConv(embeddings)
8
9  dgcl.init()
10 graph, features = ...    # Load data
11 dgcl.build_comm_info(graph, topology)
12 features = dgcl.dispatch_features(features)
13 model = distributed_gcn_model(...)
14 loss_func = ...
15
16 for i in epoch:
17   loss = loss_func(model(features))
18   loss.backward()
```

Listing 1. Integrate DGCL with DGL for GCN training

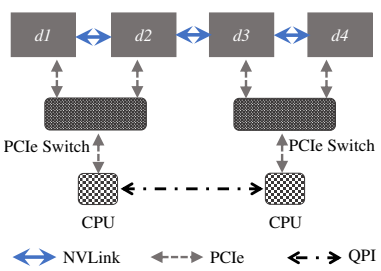
each GNN layer. The objective is to minimize the end-to-end communication time for the layer and the communication plan describes how the messages between each pair of GPUs will be transmitted. Communication plans are constructed before training starts and issued to the DGCL clients. When training starts, the clients coordinate with each other using progress flags to carry out the communication plan.

4.2 API

DGCL exposes a user-friendly API to extend a single-GPU GNN system to distributed training. We highlight the functions in the API that are tailored for GNN training as follows:

- *init()* initializes the distributed communication environment, and sets up the connections between DGCL master and clients.
- *buildCommInfo(graph, topology)* partitions a graph to a set of GPUs, builds the communication relation among the GPUs and runs the communication planning algorithm to obtain the communication strategy.
- *embeddings graphAllgather(localEmbeddings)* fetches the embeddings of the remote vertices of a GPU and sends the embeddings required by other GPUs according to the communication relation. The returned value *embeddings* contains all the embeddings (i.e., both local and remote) for executing a GNN layer on the GPU.

We call the operation that passes vertex embeddings among GPUs *graph Allgather* as its semantics is similar to *Allgather* operation in collective communication, i.e., after applying the operation, every worker gets its required data. For example, in the graph partitioning in Figure 1b, after *graph Allgather*, GPU 1 will have the embeddings of vertex $\{a, b, c, d, f, j\}$. *graph Allgather* is a synchronous operation, DGCL blocks a client until it receives all the remote vertex embeddings.



	$d3$	$d4$
$d1$	$\{a, b, c, d\}$	$\{a, b, e, f\}$
$d2$	$\{g, h, i\}$	$\{g, i, k, l\}$

Figure 6. An example of communication topology and communication relation, in which d_1 and d_2 are senders

DGCL can be easily combined with existing GNN systems with the API presented above and we show an example in List 1, which uses DGCL in DGL [35] to train a GCN model using our Python API. Lines 9-12 initialize the communication environment, conduct communication planning and dispatch the graph partitions to their corresponding GPUs. Line 6 invokes *graph Allgather* to collect the remote embeddings and Line 7 conducts DGL graph convolution in the single-GPU mode.

5 Communication Planning

In this section, we first formulate the communication planning problem by introducing the communication strategies for each vertex and the cost model. Then we present the shortest path spanning tree (SPST) algorithm, which solves the communication planning problem in a greedy manner.

5.1 Problem Formulation

Communication planning takes the *communication relation* and *communication topology* as input, and tries to minimize the communication time for a GNN layer. As introduced in §4, the communication relation is modeled by the (d_i, d_j, V_{ij}) tuples, where V_{ij} is the vertex embeddings GPU d_i needs to send to GPU d_j. The communication topology D describes the connections among the GPUs and each physical connection $l_j \in D$ is associated with a bandwidth b_j. We provide an example of communication relation and communication topology in Figure 6, and discuss some considerations in GNN communication planning using Figure 6 as follows.

- **Utilize fast links**: To send data from d_1 to d_3, we may either (1) go through the slow PCIe-QPI-PCIe direct link, or (2) transfer the data to d_2 first and then forward to d_3 via fast NVLink. According to our analysis in §3, (2) may be a better choice and thus planning should allow multi-hop forwarding to fully utilize the fast links.
- **Fuse communication**: Vertices $\{a, b\}$ on d_1 are required by both d_3 and d_4, and thus sending $\{a, b\}$ to d_3 and d_4 separately may be less efficient than sending $\{a, b\}$ to d_3 first and allowing d_3 to forward to d_4. Thus,

for vertices required by more than one remote GPUs, communication fusion should be supported.

- **Avoid contention**: d_1 sending data to d_3 and d_2 sending data to d_4 may happen concurrently on the same direct PCIe-QPI-PCIe link, causing contention and reducing the attainable bandwidth of each other as discussed in §3. Planning should avoid or consider the influence of these contentions.
- **Balance loads on different links**: d_1 may send some of the vertices to d_3 via direct PCIe-QPI-PCIe link and some are forwarded by d_2. As putting too much load on one link may create stragglers, it is crucial to balance the loads on different links.

Feasible communication strategies. For a vertex u, denote its source GPU as s_u (i.e., u belongs to the partition of s_u) and the set of remote GPUs that need u's embedding as D_u. Among the different strategies to transfer u's embedding from s_u to each GPU in D_u, we discuss which strategies need to be considered in our communication planning algorithm. Apparently, the optimal communication strategy for a vertex u is a tree computed from the communication topology D, which is rooted at s_u and contains all GPUs in D_u. This can be trivially proved by contradiction: if the optimal communication strategy for u is not a tree, we can construct a spanning tree of the optimal strategy by removing some edges (i.e., eliminating some communication) and the spanning tree will not have larger communication cost than the optimal strategy. Thus, for each $u \in V$, we only need to consider communication strategies that are trees rooted at s_u. We call these strategies the set of *feasible communication strategies*.

Note that we define the communication strategy for each vertex $u \in V$ individually. This means that we allow vertices from the same source GPU to use different communication strategies, which has the following benefits. First, it provides flexibility for *load balancing*, e.g., d_1 can send some vertices via NVLink and the others via PCIe. Second, it allows *multi-hop forwarding* for a single vertex. For example, in Figure 6, $T_a : d_1 - d_2 - d_3 - d_4$ uses d_2 to forward the embedding of a to d_3 and d_4, and uses fast NVLink all the way.

Cost model. Let T_u be a feasible communication strategy of a vertex $u \in V$. We define a *communication plan* S for a GNN layer as $S = \cup_{u \in V} T_u$. We also define a *cost model* $t(S)$ for a plan S, which measures the *communication time* needed using S for distributed GNN training. We use $t(S)$ to guide the search for the optimal strategies. We assume that the communications in S happen in *stages* and the stage of a communication is determined by the number of links from the source GPU. Given the communication trees of all vertices (i.e., a communication plan), calculating the cost takes three steps: (1) calculating the amount of data transfer on each link in each stage; (2) calculating the communication time of each stage by taking the maximum communication time of the active links in the stage; (3) calculating the total communication time as the sum of the stages. For example, in Figure 6, for vertex a and $T_a : d_1 - d_2 - d_3 - d_4$, we have $d_1 - d_2$, $d_2 - d_3$ and $d_3 - d_4$ conducted in stage 1, 2 and 3, respectively. We also assume that the communications in the same stage happen concurrently and a stage finishes only when all its communications complete. For example, if $T_a : d_1 - d_2 - d_3 - d_4$ and $T_g : d_2 - d_3 - d_4$, then d_1 sends a to d_2 and d_2 sends g to d_3 concurrently as they are both in stage 1.

Dividing the communications into stages allows to batch the transfer of different vertices to fully utilize the bandwidth. For example, if $T_a = T_b : d_1 - d_2 - d_3 - d_4$, then a and b are sent from d_1 to d_2 together on NVLink via one communication operation. In addition, communications in the same stage are parallelized on different links. Note that we call the connection between two computing GPUs (or workers) a link. The link may have only one physical connection, e.g., the NVLink between d_1 and d_2 in Figure 6, or multiple physical connections, the PCIe-QPI-PCIe link from d_1 to d_3. We use L to denote the set of all links in the topology D.

We apply the following rules to calculate $t(S)$:

- For a direct link $L_i \in L$ between the GPUs, e.g., an NVLink, its communication time in stage k is $t_i^k = c_i^k(S)/b_i$, in which $c_i^k(S)$ is the amount of data transfer in S that happens on link L_i at stage k, and b_i is the bandwidth of link L_i.
- For a link $L_i \in L$ that contains multiple physical connections, e.g., the PCIe-QPI-PCIe link between d_1 and d_3, we decompose it into multiple physical hops L_{ij}, and have $t_i^k = \max_j t_{ij}^k$, in which $t_{ij}^k = c_{ij}^k(S)/b_{ij}$ and $c_{ij}^k(S)$ is the aggregate data transfer that goes through physical hop L_{ij} at stage k.
- The communication time of a stage k is $t^k = \max_i t_i^k$. The total communication time of a plan S is $t(S) = \sum_{k=1}^{m-1} t^k$, where we assume that D contains m GPUs and hence there are at most $m - 1$ stages. [2]

In our cost function, using the maximum communication time of the physical connections as the communication time for a link allows to account for contention. For example, when d_1 sends $\{a, b\}$ to d_3, and d_2 sends $\{g, h\}$ to d_3 via the PCIe-QPI-PCIe link in Figure 6 in the same stage, the communication time will be decided by transferring $\{a, b, g, h\}$ (i.e., the aggregate data) over the QPI connection between the CPUs. This assumption is also reasonable as the communications on multiple physical connections are pipelined and thus the communication time depends on the slowest connection. Defining the stage communication time as the maximum among the links allows us to balance the load

[2] This is because the communication strategy of each vertex is a tree.

Algorithm 1 Shortest Path Spanning Tree (SPST)

1: **Input:** The communication topology graph $D(V', E')$, the source GPU s_u and the destination GPUs D_u for each vertex $u \in V$.
2: **Output:** The communication strategies for all vertices in V.
3: **for** $u \in V$ **do**
4: $N_{src} = \{s_u\}$
5: **for** i from 1 to $|D_u|$ **do**
6: $C = incrmentalLinkCost(D, S)$ ▷ *Use Algorithm 2*
7: Run $dijkstra(N_{src}, D, C, D_u \backslash N_{src})$
8: Let p be the shortest path among the set of shortest paths from each $d \in N_{src}$ to each $d' \in D_u \backslash N_{src}$ computed in Dijkstra algorithm
9: $N_{src} = N_{src} \cup \{$ vertices on $p\}$
10: $S = S \cup \{$ edges on $p\}$
11: **end for**
12: **end for**
13: **Return:** S

among different links because putting too much load on one link will result in a long stage communication time.

We define the communication planning problem as follows:

$$\begin{gathered} \min t(\cup_{u \in V} T_u) \\ \text{s.t. } T_u \in D \text{ and is a tree that contains } s_u \text{ and } D_u. \end{gathered} \tag{2}$$

An interesting property of our communication planning problem is that *the optimal plan is irrelevant to the feature dimension of the vertex embeddings*. Adjusting the feature dimension creates a common linear scaling on the communication time on all links and stages, and thus an optimal plan for a feature dimension f is also optimal for another feature dimension f'. A corollary from this property is that the same optimal communication plan applies for different GNN layers and models as they share the same communication relation and may differ only in embedding dimension.

5.2 The SPST Algorithm

For each vertex $u \in V$, communication planning needs to find a rooted spanning tree T_u of the communication topology D as its communication strategy. T_u should use the source GPU of u (i.e., s_u) as root and contain all destination GPUs in D_u. We use a shortest path spanning tree (SPST) algorithm for communication planning, which is presented in Algorithm 1.

SPST computes the communication strategy for the vertices one by one, and we randomly shuffle the vertices of $G = (V, E)$ before execution. When processing a vertex u, SPST considers the communication strategies of the processed vertices as fixed, and minimizes the increase in the overall communication time brought by u. For a vertex u, SPST keeps a set of source GPUs N_{src}, which is initialized as s_u and records the GPUs that have received u. In each

Algorithm 2 incrementalLinkCost

1: **Input:** The communication topology graph $D(V', E')$, and a set of communication strategies S.
2: **Output:** Two dimensional array with shape $(|V'|, |E'|)$ modeling the cost of using different links at different stages.
3: $C = array2D(|V'|, |E'|)$
4: **for** $e_j \in |E'|$ **do**
5: **for** i **from** 0 **to** $|V'| - 1$ **do**
6: $C(i, e_j) = costFunc(S \cup \{(i, e_j)\}) - costFunc(S)$
7: **end for**
8: **end for**
9: **Return:** C

iteration (Lines 5-11), SPST invokes a multi-source shortest path algorithm (Line 7) to find the shortest path from each GPU in N_{src} to each GPU in $D_u \backslash N_{src}$. Then it picks the path p that has the shortest cost distance among all the paths found in Line 8, and adds the GPUs on p to N_{src} (Line 9) and the links on p to the cumulative communication plan S (Line 10). As one destination GPU is added to N_{src} in each iteration, SPST includes all GPUs in D_u in the spanning tree after $|D_u|$ iterations and thus finishes processing vertex u.

Line 6 of Algorithm 1 invokes Algorithm 2 to calculate the edge weight matrix C for path finding, where $costFunc(\cdot)$ is the cost model discussed in §5.1 and $C(i, e_j)$ is the increase in the overall communication time when transferring a vertex embedding on link e_j at stage i. The cost matrix C has $|V'|$ rows as for a communication topology with $|V'|$ nodes, a spanning tree (i.e., T_u) has at most $|V'|$ stages (otherwise there will be at least one circle). Algorithm 1 uses $C(i, e_j)$ as the edge weight for link e_j if e_j is i-hops away from the source GPU s_u in the communication spanning tree T_u for vertex u. In each iteration of the SPST algorithm, the costs of the edges on the same path p can be added because these edges belong to different stages.

There are several intuitions behind our SPST algorithm:

- **Load balancing**: SPST minimizes the blow-up in the overall communication time when propagating a vertex to a destination GPU. This automatically balances the load on different links because for an under-loaded link L_i with $t_i^k < t^k$ (i.e., the link communication time is smaller than the stage time), adding communication on it will not increase the overall communication time.
- **Prioritizing fast links and forwarding**: SPST expands the spanning tree to the "nearest" destination GPU with the smallest time cost in each step. Thus, SPST prefers fast links (e.g., NVLink) over slow links (e.g., QPI) as fast links have a small communication cost. The sequential tree expansion also encourages using intermediate GPU to forward vertices to multiple destinations, i.e., transferring to one destination GPU first and forwarding to the others via fast links.

- **Avoiding contention and parallelizing communication**: SPST finds the shortest path from N_{src} to one destination GPU each time and hence the links on the path are in different stages. [3] As we conduct the stages sequentially, the communications in a path are contention-free and their costs are addable. The cost function of SPST assumes that the communications on different links can happen in parallel in the same stage. Using other algorithms may require more sophisticated handling of contention and parallelization.

Readers may wonder whether finding the optimal communication rooted tree for a vertex can be modeled as a steiner tree problem [37], which is NP-hard but there are known approximate solutions. The answer is negative because the communication costs for the same vertex are not addable as different links (for different paths) in the rooted tree can be activated in the same stage. This is also why we need to update the cost matrix C in each of the $|D_u|$ iterations of Algorithm 1 when processing a vertex u.

Complexity analysis. Each vertex u in $G = (V, E)$ needs to run the multi-source shortest path algorithm at most $O(|D_u|) = O(|V'|)$ times, where $|V'|$ is the number of GPUs. The shortest path algorithm has a complexity of $O(|E'| \log |E'|)$, where $|E'|$ is the number of links in the communication topology graph $D(V', E')$. Therefore, the overall complexity of SPST for all vertices is $O(|V||V'||E'| \log |E'|)$. The complexity is linear w.r.t. the number of vertices in the data graph G and not high because both $|V'|$ and $|E'|$ are small (e.g., < 100) for typical distributed training scenarios.

Readers may notice that if we use Algorithm 2 to calculate the cost of all links and stages, the complexity of the shortest path algorithm is at least $O(|E'||V'|)$, which may be larger than $O(|E'| \log |E'|)$. In this case, we can still maintain the $O(|E'| \log |E'|)$ complexity by calculating the costs of the links in an on-demand fashion. However, this optimization will make the algorithm more complex but the main idea remains the same, and thus we omit it in the paper.

6 System Design and Implementation

In this section, we discuss the system designs in DGCL and give some implementation details.

6.1 Decentralized Communication Coordination

The communication plan S produced by the SPST algorithm is organized into $(d_i, d_j, k, T^s_{ij}, T^r_{ij})$ tuples for execution. In a tuple, d_j is a GPU that has link with d_i, k is the stage number, T^s_{ij} and T^r_{ij} (called the *send*/*receive* table) contain the vertex ids of the embeddings d_i needs to send to and receive from d_j at stage k, respectively. This organization batches the communication of different vertices passing through the same pair of GPUs and helps fully utilize the network bandwidth. In the backward pass, stages are executed in a reverse order, and T^s_{ij} and T^r_{ij} are switched as gradient flow in the opposite direction of embeddings. The $(d_i, d_j, k, T^s_{ij}, T^r_{ij})$ tuples are issued to the GPUs before training starts and do not take too much memory as T^s_{ij} and T^r_{ij} contain vertex ids instead of high-dimensional vertex embeddings. Moreover, the same communication tuples are reused for all GNN layers.

To execute the communication plan, a natural solution is *centralized coordination*, in which each GPU notifies the DGCL master when it finishes a stage and blocks until the master tells it to start the next stage. However, centralized coordination may have a high overhead for communicating with the DGCL master and waiting for stragglers. Therefore, we use a decentralized communication coordination protocol based on *ready* and *done* flags. When a GPU is ready for communication in a stage, it sets its *ready* flag to be true and waits for the *ready* flags of its peer GPUs. When one of its peers becomes ready, the GPU starts to send data to that peer. Once all data have been sent to the buffer of the peer GPU, it sets its *done* flag for that peer to be true so that the peer GPU can retrieve data from the buffer. After sending and retrieving all required data, a GPU becomes ready for the next stage. The flags of a GPU can be accessed by its peer GPUs directly, which alleviates the communication bottleneck on the master. In addition, transient stragglers in communication will not block the other GPUs.

6.2 Efficient Communication Kernel

Automatic communication method selection for GPUs. DGCL uses different peer-to-peer communication strategies when two GPUs have different connections. (1) For a pair of GPU under the same CPU socket, DGCL conducts communication using the CUDA virtual memory technique. The sender gets the addresses of the data buffer and the *ready* flag on the receiver from the driver, and the receiver gets the address of the *done* flag of the sender. The sender and receiver coordinate by setting the flags. (2) For GPUs under different CPU sockets, DGCL conducts communication through pinned CPU memory as it has better performance than CUDA virtual memory in this case. A shared memory buffer is allocated on the CPU, and the sender and receiver access the memory address using direct memory access (DMA). (3) For a pair of GPUs that reside on different machines, an extra thread is assigned to help data send/receive through NIC (e.g., Ethernet or IB). The sender moves the data to a local buffer first and the helping thread sends the data to the remote machine. Once the data has been sent, the helping thread sets the flag on the local machine as ready. GPU RDMA is utilized if IB and the CUDA version supports it.

Non-atomic update in back-propagation. In the backward pass, a vertex embedding will receive gradients from multiple GPUs if the vertex is used as a remote vertex by these GPUs. As the gradients from different GPUs are received by different CUDA threads, atomic reduction is needed

[3] This is because the edges on the path have different tree depths

due to potential data conflict, which incurs performance overhead. To remove the overhead, DGCL divides a stage into several sub-stages such that a vertex receives gradients from only one GPU in each sub-stage. We support sub-stage communication without changing the planning algorithm. A planned communication tuple $(d_i, d_j, k, T^s_{ij}, T^r_{ij})$ is divided into $|D|-1$ smaller tuples $(d_i, d_j, k, l, T^s_{ij}(l), T^r_{ij}(l))$, and the *receive* table T^r_{ij} is partitioned into $|D|-1$ parts (i.e., the $T^s_{ij}(l)$ sub-tables) such that gradients for the same vertex from different GPUs are in different sub-stages. The *send* table is adjusted according to the *receive* table.

Data packing. DGCL packs the vertex embeddings into 16 bytes to improve efficiency. 16 bytes is the maximum data size that one CUDA thread can fetch in one instruction and packing reduces the number of memory accesses.

6.3 Implementation Details

In DGCL , we use multiple processes for computation and maps one computation process to each GPU. A master process is utilized for coordination. The initialization of the distributed communication environment depends on the parallel execution framework, e.g., MPI or PyTorch distributed package. For MPI, the master process generates a communication address and broadcasts it to the other processes. For PyTorch distributed package, multiple processes are spawn from one process and the processes can read the address of the master from the environment variables. After all processes have connected with the master process, the master uses *gather* and *scatter* for distributed training, e.g., assign the partitioned sub-graphs, dispatch vertex features and exchange GPU connection information. DGCL leverages existing data parallel frameworks such as Horovod and PyTorch distributed data parallel package for distributed model synchronization. As the model size is usually small for GNNs, we do not conduct optimizations for it.

In existing GNN systems, execution is decomposed into two parts, i.e., graph relevant operations such as graph aggregation and standard DNN operations such as matrix multiplication. On a single machine, the graph relevant operations need to involve the remote vertices while the DNN operations do not need to compute the embeddings for the remote vertices. Therefore, we use *graphAllgather* to collect the remote vertices before the single machine GNN system conducts the graph operations. After the graph operations, we remove the embeddings of the remote vertices before the single machine GNN system conducts the normal DNN operations. As a result, we do not incur extra computation overhead for distributed training.

7 Experimental Evaluation

We evaluated our method with two hardware configurations. The default configuration contains two servers, each equipped with 8 V100 GPUs with 16 GB memory and the connections among the GPUs on the same machine follows Figure 3. The GPUs on one machine communicate with peers on the other machine using the same IB NIC card and GPU RDMA [23] is enabled for GPUs under the same PCIe switch with the IB NIC card. The second configuration is a server that contains 8 1080-Ti GPUs with 12 GB memory. The GPUs are connected using PCIe instead of NVLink.

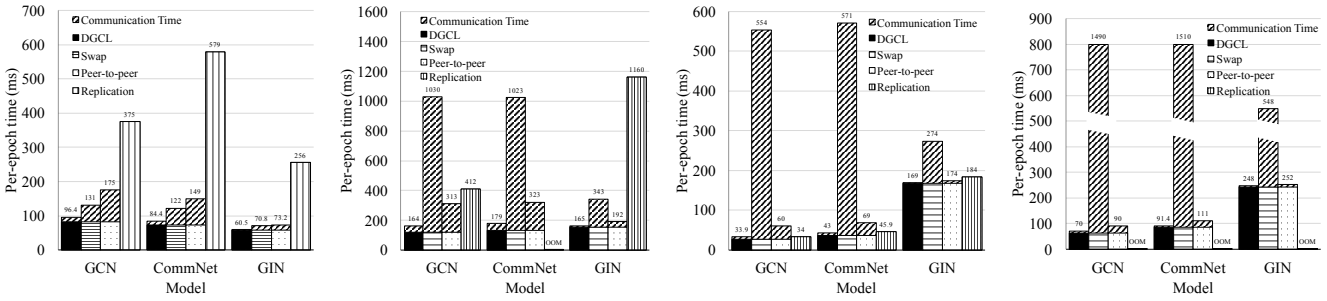
Table 4. Dataset statistics and model configurations

	Reddit	Com-Orkut	Web-Google	Wiki-Talk
Vertices	0.23M	3.07M	0.87M	2.39M
Edges	110M	117M	5.1M	5.0M
Avg. Degree	478	38.1	5.86	2.09
Feature Size	602	128	256	256
Hidden Size	256	128	256	256

Datasets. Table 4 lists the statistics of the graphs used in the experiments. Reddit [8] is a post-to-post graph, where vertices are posts and an edge connects two posts if there is a user that comments on both of them. Com-Orkut [41] models a social network, in which vertices are users and edges represent the friendship relation between users. In Web-Google [19], the vertices are web pages and the edges are the hyperlinks between the web pages. Wiki-Talk [18] records user interactions on Wikipedia, in which vertices are Wikipdia users and an edge from user u to user v means that u edited the talk pages of v at least once. Among the 4 graphs, Reddit and Com-Orkut are relatively dense, while Web-Google and Wiki-Talk are sparse. Com-Orkut and Wiki-Talk contain more than 2 million vertices, while Reddit and Web-Google are smaller. We choose graphs with different properties to validate DGCL under different settings.

GNN models. We used three popular GNN models. *GCN* [17] is widely used for semi-supervised learning on graphs and aggregates neighbor embeddings using simple weighted sum. *CommNet* [32] models multiple cooperating agents, which learn to communicate among themselves before taking actions. *GIN* [39] uses multi-layer perceptions (MLPs) to update the vertex embedding and is shown to match the powerful Weisfeiler-Lehman graph isomorphism test. From GCN to CommNet and GIN, the models have an increasing computation complexity, and we used them to explore the interplay between the computation and communication costs. We used 2 layers for all GNN models as 2-layer GNNs are the most popular. The dimensions of the input feature and hidden feature can be found in Table 4. For graphs that do not come with vertex features, we randomly generate the 0-th layer vertex embeddings.

Baselines and evaluation methodology. A direct comparison with NeuGraph [21] and ROC [13] is not feasible because they are not yet open-source. We compared *DGCL* with



(a) Reddit **(b)** Com-Orkut **(c)** Web-Google **(d)** Wiki-Talk

Figure 7. The per-epoch time and communication time for training the 3 GNN models on 4 datasets with 8 GPUs

three baselines, *Swap*, *Peer-to-peer* and *Replication*. *DGCL*, *Swap* and *Peer-to-peer* use the same non-overlapping graph partitioning to assign vertices in a graph to the GPUs. *DGCL* uses the DGCL library for embedding passing. Following ROC, *Peer-to-Peer* uses direct peer-to-peer communication. *Swap* follows NeuGraph [21] by dumping vertex embeddings to main memory after each layer for communication, and we also implemented the chain-transfer optimization in NeuGraph for high efficiency. *Replication* eliminates embedding passing by replicating vertices as discussed in §3. For fair comparison, all methods used DGL for single-GPU execution.

Our main performance metrics are *per-epoch time* and *communication time*. Per-epoch time is the time to conduct a forward and backward pass for all vertices in the graph. As all our baselines are equivalent in single-GPU training from the algorithm perspective, shorter per-epoch time means better time-to-accuracy performance. Communication time is the time used to conduct embedding passing in an epoch. All reported timing results are measured after warm-up and averaged over 10 repetitions.

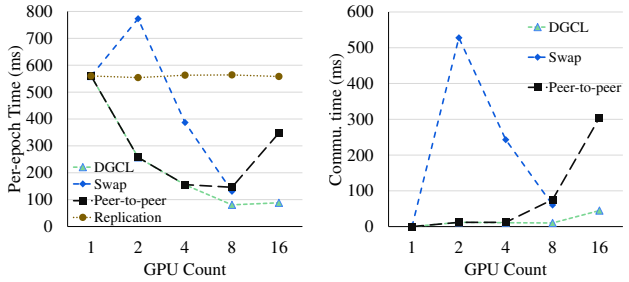
7.1 Main Results

We report the per-epoch time and communication time for training the three GNN models on the four datasets with 8 GPUs in Figure 7. We decompose the per-epoch time into communication time and computation time, and plot the communication time at the top of each bar. Note that *Replication* has zero communication time as it does not need to conduct communication. We can make the following observations: (1) *Replication* has severe performance penalty. Its per-epoch time is significantly longer than the other methods for the dense Reddit graph (see Figure 7a) and it goes OOM for the larger Com-Orkut and Wiki-Talk graphs. However, for the small and sparse Web-Google graph, *Replication* outperforms *Peer-to-peer* and *Swap* because a relatively small number of vertices are replicated. (2) *Swap* has the worst performance on the three larger graphs, Com-Orkut, Web-Google and Wiki-Talk, as it needs to swap all vertex embeddings to main memory. However, *Swap* performs well for the small and dense Reddit graph because a large portion of the vertices swapped to main memory are required by remote GPUs. (3) *Peer-to-peer* has a better overall performance than *Replication* and *Swap*.

We also have the following observations for *DGCL*. (1) It has significantly shorter communication time than *Peer-to-peer* and *Swap*. The communication time of *Peer-to-peer* is up to 7.04x and on average 4.45x of *DGCL*'s time, while that of *Swap* is up to 230x and on average 60.7x of *DGCL*'s time. (2) As a result of its short communication time, *DGCL* achieves the shortest per-epoch time among the four methods in all cases. Averaged across all graphs and models, the per-epoch time of *Peer-to-peer* and *Swap* are 1.47x and 7.43x of *DGCL*'s time. This result is remarkable as for sparse graphs (i.e., Web-Google and Wiki-Talk) or complex models (e.g., GIN), computation dominates the execution time and *DGCL* needs to spend the same computation time as *Peer-to-peer* and *Swap* as they all use DGL for single-GPU execution. (3) *DGCL* has the most significant performance gain for training GCN on Com-Orkut, in which case the per-epoch time of the baselines is reduced by at least 47.7%. Overall, *DGCL* is most efficient for dense graphs and simple models, for which the communication time takes up a large portion of the per-epoch time.

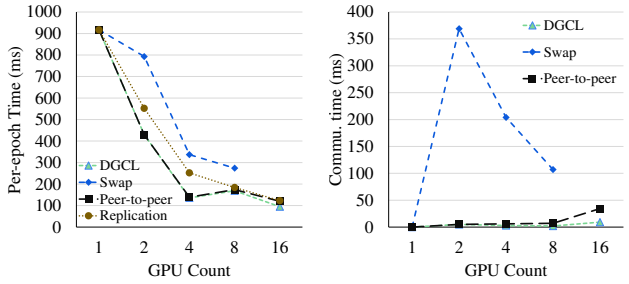
To show the generality of DGCL when using different number of GPUs, we report the results of training GCN on Reddit in Figure 8 and GIN on Web-Google in Figure 9. We choose the two smaller graphs as a small number of GPUs do not have enough memory to process the larger graphs. For the same reason, we do not report GIN on Web-Google using 1 GPU. Also, as *Swap* is designed for a single machine in NeuGraph, we do not use it for 16 GPUs with two servers.

The results show that *DGCL* consistently achieves the shortest per-epoch time among all methods and has shorter communication time than both *Swap* and *Peer-to-peer*. The methods have similar per-epoch time for training GIN on Web-Google because the computation time dominates the per-epoch time due to the complex model. The communication time of *Swap* drops when increasing from 2 GPUs to 8 GPUs because there are more GPUs to dump/load the vertex embeddings from/to main memory in parallel. *Peer-to-peer* and *DGCL* have identical communication time when



(a) Per-epoch time **(b)** Communication time

Figure 8. Per-epoch time and communication time for training GCN on Reddit with different number of GPUs



(a) Per-epoch time **(b)** Communication time

Figure 9. Per-epoch time and communication time for training GIN on Web-Google with different number of GPUs

Table 5. Per-epoch time (ms) for training GIN and GCN on 16 GPUs, where *DGCL-R* used replication to eliminate cross-machine communication

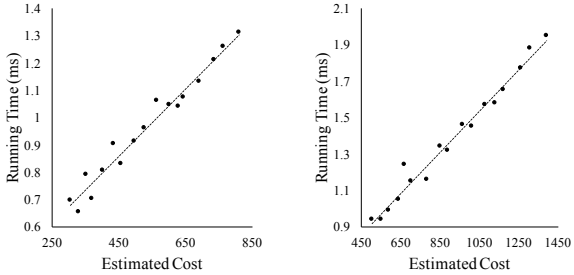
	Web-Google		Reddit	
	DGCL	DGCL-R	DGCL	DGCL-R
GCN	54.0	26.7	88.4	86.4
GIN	94.8	107	53.1	71.9

there are 4 or fewer GPUs because these GPUs have direct NVLink connections among each other, and both methods use NVLink. When the connections are more complex, e.g., with 8 or 16 GPUs, *DGCL* has significantly shorter communication time than *Swap* and *Peer-to-peer*. For training GCN on Reddit with 16 GPUs, the per-epoch time of *Peer-to-peer* and *Replication* are 3.94x and 6.31x of *DGCL*'s time.

Figure 8 and Figure 9 show that distributed GNN training does not scale well with 16 GPUs due to slow inter-machine communication with IB. Thus, we introduce an alternative scheme, i.e., *DGCL-R*, which replicates vertices to eliminate inter-machine communication as in *Replication* and uses *DGCL* to plan communication for GPUs in the same machine. Table 5 reports the per-epoch time of *DGCL-R* and *DGCL* for training GCN and GIN on Web-Google and Reddit. The results show that *DGCL-R* has longer per-epoch time than

Table 6. Time (ms) for one *graphAllgather* operation in a hardware configuration without NVLink

	Reddit	Com-Orkut	Web-Google	Wiki-Talk
DGCL	14.3	128	7.84	5.86
Swap	14.5	1220	116	317
Peer-to-peer	17.9	179	8.72	8.51



(a) Web-Google **(b)** Reddit

Figure 10. Relation between the model estimated communication cost and the actual communication time for one *graphAllgather* operation with 8 GPUs

DGCL when training GIN because GIN is computation intensive and replication introduces extra computation. *DGCL-R* also does not perform well for Reddit, a more dense graph, as it replicates almost the entire graph on each machine. However, *DGCL-R* significantly outperforms *DGCL* for training GCN on Web-Google because the graph is sparse and inter-machine communication dominates per-epoch time for the simple GCN model.

To show that *DGCL* can cope with different hardware connections, we report the time taken by one *graphAllgather* operation in our second hardware configuration without NVLink. Recall that *graphAllgather* collects the remote vertex embeddings for each GPU for training a GNN layer and hence reflects the communication time. The feature size is 128 and there are 8 GPUs. Table 6 shows that *graphAllgather* uses less time with *DGCL* than with both *Swap* and *Peer-to-peer*. *Swap* takes a long time for the large graphs (i.e., Web-Goggle, Com-Orkut and Wiki-Talk), which is consistent with its poor performance for these graphs in Figure 7. *DGCL*'s advantage over *Peer-to-peer* is less significant compared with Figure 7 as this configuration does not have NVLink. *DGCL* outperforms *Peer-to-peer* in this case mainly because it considers contention and load balancing.

7.2 Micro Benchmarks

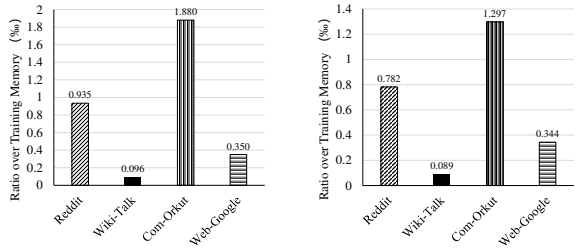
As we use the cost model in §5.1 to guide communication planing, it is crucial that the model estimated communication cost is an accurate estimate of the actual communication time. To test the accuracy of our cost model, we plot the estimated cost and actual communication time of one graphAllgather operation on Reddit and Web-Google in Figure 10.

Table 7. The breakdown of the communication time (ms) of one *graphAllgather* operation for DGCL with 8 GPUs

	NVLink	Others	Relative difference
Web-Google	0.787	0.821	4.32%
Reddit	1.16	1.07	7.41%
Com-Orkut	7.43	7.30	1.78%
Wiki-Talk	0.783	0.882	12.6%

Table 8. Running time (s) of SPST (s)

	Reddit	Com-Orkut	Web-Google	Wiki-Talk
2 GPU	0.74	4.61	0.78	0.37
4 GPU	1.52	16.2	1.56	0.65
8 GPU	4.19	43.5	3.42	1.45
16 GPU	9.91	110	6.76	3.14



(a) 8 GPU **(b)** 16 GPU

Figure 11. The ratio between the memory used for the *send*/*receive* tables and normal GNN training

The communication volume (hence communication time) is controlled by communicating only some vertices in the original graph. The results show that the actual communication time has a linear relation with the estimated communication cost. We fitted a line between them and found that the divergence from the line is below 5% in most cases.

One important goal of the SPST algorithm is to balance the communication time of different links to avoid communication straggler. We give the breakdown of the communication time that DGCL spends on different links in Table 7. To avoid interference, the communication time of NVLink is measured after removing the communication on other links and vice versa. The results show that DGCL can effectively balance the communication time spent on different links.

We report the running time of the SPST algorithm for different graphs and number of GPUs in Table 8, which is measured on a machine with a 48-core Platinum 8160 CPU and 256G main memory in the single-thread mode. The results show that the SPST algorithm can finish communication planning in several seconds in most cases. The trend is that SPST takes more time when the graph is large (i.e., more vertices to process) or dense (i.e., more multi-source shortest-path iterations for each vertex). In addition, the running time grows approximately linearly with the number of GPUs.

Table 9. Time (ms) for *graphAllgather* in the backward pass

	Reddit	Com-Orkut	Web-Google	Wiki-Talk
Atomic	1.72	14.3	1.11	0.99
Non-atomic	1.28	9.16	0.83	0.71

We report the ratio between the memory used to store the *send*/*receive* tables (i.e., T^s_{ij}, T^r_{ij}) for decentralized communication coordination and the peak memory consumption during normal GNN training in Figure 11. The results show that the ratio is below 0.002 in all cases, indicating that decentralized communication coordination has a low memory overhead. This is because we store only vertex ids instead of high-dimensional vertex embeddings and the same *send*/*receive* tables are reused for different GNN layers. To validate the effectiveness of non-atomic aggregation in back propagation, we report the time of one *graphAllgather* operation in the backward pass in Table 9. We used 8 GPUs in the default configuration with NVLink and the dimension of the hidden layer is 128. The results show that non-atomic aggregation effectively reduces the running time of atomic aggregation.

8 Related Work

In this section, we review related works on GNN systems and communication planning.

8.1 Existing GNN Systems

DGL [35] and PyG [7] integrate with existing deep learning frameworks (e.g., Tensorflow [1] and PyTorch [27]) and provide sparse tensor operations tailored for graph data. They consider training on a single GPU and currently do not support full graph training on multiple GPUs. Due to the heavy computation workload of GNN training, using a single GPU may result in a long per-epoch time for large graphs. Moreover, a single GPU may run out-of-memory (OOM) as each vertex needs to keep the high dimensional embedding during training.

NeuGraph [21] supports GNN training on multiple GPUs in a single machine by partitioning a graph into partitions and assigning each GPU to handle some partitions. The CPU memory is used for data exchange: the GPUs fetch the required vertex embeddings from CPU memory for computation and write the embeddings back to CPU memory after finishing a layer. As the GPUs access CPU memory via relatively slow PCIe links, the I/O overhead is reported to be high, taking up about 90% of the per-epoch time in some cases. In addition, it is unclear how to extend NeuGraph to multiple machines due to the high I/O overhead.

ROC [13] also uses graph partitioning to divide the workload among the GPUs but supports training with multiple

machines. A linear regression model is learned to predict the execution time of each GPU and graph partitioning is adjusted accordingly to balance the workload of the GPUs. When GPU memory is insufficient (e.g., for large graphs), ROC uses a dynamic programming algorithm to choose the tensors to swap to CPU memory such that the data transfer is minimized. However, ROC is based on Lux [12], which uses peer-to-peer communication for vertex state exchange. We have shown that peer-to-peer communication has high overheads for distributed GNN training in §3.

Seastar [38] allows users to program GNN models easily with a vertex-centric programming model in native Python syntax. GNN models implemented with Seastar's API are intuitive to understand as the vertex-centric programming model provides a direct one-to-one mapping from the GNN formula to the implementation code. Seastar achieves excellent performance by just-in-time compiling and optimizing the vertex-centric function with Seastar operator fusion and kernel-level optimizations such as feature-adaptive computing, locality-centric execution and dynamic load balancing. However, Seastar currently only supports single-GPU GNN training. In this regard, DGCL can integrate with Seastar for scalable GNN training, where DGCL handles communication planning and execution (also other tasks such as graph partitioning) while Seastar handles the single-GPU GNN training. DGCL and Seastar can further integrate with MindSpore [11], a deep learning backend framework, to accelerate distributed training using MindSpore's features such as auto-parallel training and tensor-compilation techniques.

8.2 Communication Planning

GPU-based graph processing systems usually target at graph workloads such as PageRank, BFS and connected components. Some of them build a ring topology for cross-GPU communication [2, 42], some conduct communication via CPU memory [16], some use replication to eliminate cross-GPU communication [43], and some use direct peer-to-peer communication [26]. However, these works do not conduct fine-grained communication scheduling for multi-gpu graph processing by considering the underlying communication topology.

There are some works [6, 24] that use collective communication (e.g., *allreduce* and *allgather* operations [3, 28, 29]) for GPUs when training regular deep neural networks (DNNs). *Allreduce* aggregates the data of the same size across the GPUs and writes the result back to each GPU. *Allgather* concatenates the data from different GPUs and writes the results to each GPU. These works assume that each GPU needs to send/receive the same amount of data and the data sent by one GPU is needed by all other GPUs. These assumptions no longer hold for distributed GNN training as different GPUs need the embeddings of different vertices.

The fine-grained graph communication problem can also be formulated as a multi-source multicast routing problem (MMRP). Given the communication topology and the bandwidth of each link, the goal of MMRP is to ensure that the traffic on all links does not exceed their capacities [34], or to minimize the total traffic on all links or maximize the minimum residual bandwidth of the links [5]. However, in the GNN communication planning problem (GNN-CPP), we need to minimize the total communication time of all multicast tasks, i.e., one task for each vertex to transfer it to remote GPUs. GNN-CPP is different from MRP as we do not need to consider the bandwidth constraints of the links (thus the bandwidth limits and residual bandwidth do not apply) and the time costs of different multicast tasks are not additive due to concurrent communication (this makes it different from minimizing the total traffic).

9 Conclusions

We presented DGCL — a general and efficient communication library for distributed GNN training. By considering the properties of the embedding passing operation in distributed GNN training, DGCL designs a tailored SPST algorithm for communication planning, which jointly considers fully utilizing fast links, avoiding contention and balancing loads on different links. DGCL can be used to easily extend existing single-GPU GNN systems to distributed training with user-friendly APIs. Experimental results show that DGCL effectively reduces the time used for communication and hence improves the efficiency and scalability of distributed GNN training. We think DGCL may also benefit other distributed applications (e.g., PageRank on GPU) that has an irregular communication pattern similar to GNN training.

Acknowledgments. We thank the reviewers and the shepherd of our paper, Junfeng Yang, for their constructive comments that have helped greatly improve the quality of the paper. This work was supported by GRF 14208318 from the RGC of HKSAR.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An asynchronous multi-GPU programming model for irregular computations. *ACM SIGPLAN Notices* 52, 8 (2017), 235–248.

[3] Jehoshua Bruck and Ching-Tien Ho. 1993. Efficient global combine operations in multi-port message-passing systems. *Parallel Processing Letters* 3, 04 (1993), 335–346.

[4] Rong Chen, Jiaxin Shi, Binyu Zang, and Haibing Guan. 2014. Bipartite-oriented distributed graph partitioning for big learning. In *Proceedings of 5th Asia-Pacific Workshop on Systems*. 1–7.

[5] Yuh-Rong Chen, Sridhar Radhakrishnan, Sudarshan Dhall, and Suleyman Karabuk. 2013. On multi-stream multi-source multicast routing. *Computer Networks* 57, 15 (2013), 2916–2930.

[6] Minsik Cho, Ulrich Finkler, and David Kung. 2019. BlueConnect: Novel Hierarchical All-Reduce on Multi-tired Network for Deep Learning.

In *Proceedings of the Conference on Systems and Machine Learning (SysML)*.

[7] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[8] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[10] Mikael Henaff, Joan Bruna, and Yann LeCun. 2015. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163* (2015).

[11] Huawei. 2020. MindSpore. https://e.huawei.com/us/products/cloud-computing-dc/atlas/mindspore.

[12] Zhihao Jia, Yongkee Kwon, Galen Shipman, Pat McCormick, Mattan Erez, and Alex Aiken. 2017. A distributed multi-gpu system for fast graph processing. *Proceedings of the VLDB Endowment* 11, 3 (2017), 297–310.

[13] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems (MLSys)* (2020), 187–198.

[14] George Karypis. 1997. METIS: Unstructured graph partitioning and sparse matrix ordering system. *Technical report* (1997).

[15] George Karypis and Vipin Kumar. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing* 20, 1 (1998), 359–392.

[16] Min-Soo Kim, Kyuhyeon An, Himchan Park, Hyunseok Seo, and Jinwook Kim. 2016. Gts: A fast and scalable graph processing method based on streaming topology to gpus. In *Proceedings of the 2016 International Conference on Management of Data*. 447–461.

[17] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[18] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 1361–1370.

[19] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. 2009. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics* 6, 1 (2009), 29–123.

[20] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[21] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 443–458.

[22] NVIDIA. 2020. DGX Systems. https://www.nvidia.com/en-sg/data-center/dgx-systems. [Online; accessed 8-Oct-2020].

[23] NVIDIA. 2020. GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma. [Online; accessed 8-Oct-2020].

[24] NVIDIA. 2020. NVIDIA Collective communications library (NCCL). https://https://developer.nvidia.com/nccl. [Online; accessed 8-Oct-2020].

[25] NVIDIA. 2020. NVLink and NVSwitch. https://www.nvidia.com/en-sg/data-center/nvlink. [Online; accessed 8-Oct-2020].

[26] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D Owens. 2017. Multi-GPU graph analytics. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 479–490.

[27] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[28] Pitch Patarasuk and Xin Yuan. 2007. Bandwidth efficient all-reduce operation on tree topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8.

[29] Pitch Patarasuk and Xin Yuan. 2009. Bandwidth optimal all-reduce algorithms for clusters of workstations. *J. Parallel and Distrib. Comput.* 69, 2 (2009), 117–124.

[30] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[31] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[32] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. 2016. Learning Multiagent Communication with Backpropagation. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2244–2252. https://proceedings.neurips.cc/paper/2016/hash/55b1927fdafef39c48e5b73b5d61ea60-Abstract.html

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

[34] Chu-Fu Wang, Chun-Teng Liang, and Rong-Hong Jan. 2002. Heuristic algorithms for packing of multiple-group multicasting. *Computers & Operations Research* 29, 7 (2002), 905–924.

[35] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315* (2019).

[36] Wikipedia. 2020. InfiniBand. https://en.wikipedia.org/wiki/InfiniBand. [Online; accessed 8-Oct-2020].

[37] Wikipedia. 2020. Steiner tree problem. https://en.wikipedia.org/wiki/Steiner_tree_problem. [Online; accessed 8-Oct-2020].

[38] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: Vertex-Centric Programming for Graph Neural Networks. In *Proceedings of the Fourteenth EuroSys Conference 2021, April 26-28, 2021*. ACM.

[39] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[40] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3165–3166.

[41] Jaewon Yang and Jure Leskovec. 2015. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems* 42, 1 (2015), 181–213.

[42] Yu Zhang, Xiaofei Liao, Hai Jin, Bingsheng He, Haikun Liu, and Lin Gu. 2019. DiGraph: An efficient path-based iterative directed graph processing system on multiple GPUs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 601–614.

[43] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.