WholeGraph: A Fast Graph Neural Network Training Framework with Multi-GPU Distributed Shared Memory Architecture

1st Dongxu Yang
NVIDIA
Beijing, China
dongxuy@nvidia.com

2nd Junhong Liu
NVIDIA
Beijing, China
junliu@nvidia.com

3rd Jiaxing Qi
NVIDIA
Beijing, China
jqi@nvidia.com

4th Junjie Lai
NVIDIA
Beijing, China
julienl@nvidia.com

SC22: International Conference for High Performance Computing, Networking, Storage and Analysis | 978-1-6654-5444-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/SC41404.2022.00059

***Abstract*—Graph neural networks (GNNs) are prevalent to deal with graph-structured datasets, encoding graph data into low dimensional vectors. In this paper, we present a fast training graph neural network framework, i.e., WholeGraph, based on a multi-GPU distributed shared memory architecture. WholeGraph consists of partitioning the graph and corresponding node or edge features to multi-GPUs, eliminating the bottleneck of communication between CPU and GPUs during the training process. And the communication between different GPUs is implemented by GPUDirect Peer-to-Peer (P2P) memory access technology. Furthermore, WholeGraph provides several optimized computing operators. Our evaluations show that on large-scale graphs WholeGraph outperforms state-of-the-art GNN frameworks, such as Deep Graph Library (DGL) and Pytorch Geometric (PyG). The speedups of WholeGraph are up to 57.32x and 242.98x compared with DGL and PyG on a single machine multi-GPU node, respectively. The ratio of GPU utilization can sustain above 95% during GNN training process.**

***Index Terms*—graph neural network, GPU, shared memory architecture**

I. Introduction

In the last few years, Machine Learning (ML) has solved many challenging problems in a wide range of domains, such as image recognition [1], natural language processing [2], [3], and automatic speech recognition [4]. Convolution neural networks (CNNs) have remarkable success in image recognition based on learning from spatial data between nearby pixels. And there is a lot of work that aims to optimize convolution computing [5]–[8]. Recursive Neural Networks (RNNs) are fit for temporal sequence signals [9]. Recently, Graph Neural Networks (GNNs) are able to learn relationships from graph-structured data [10]–[12], which can represent social networks, citation networks, molecular structures, brain connections, and protein-protein interactions. Many applications such as recommender systems [13], [14], knowledge graphs [15], fraud detection [16], and drug discovery [17]–[19] have used GNNs to help them accomplish their tasks.

GNNs aim to learn node, edge, or graph representations (low dimensional vector embeddings) from irregular spatial structure, i.e., graph data. GNNs take the node or edge features from input graphs and import these features to various neural networks layer by layer, which is different from graph embedding system [20]. Each node needs to collect information from its neighbors to compute its output features during the training process of GNNs. Thus, leveraging the output features of the nodes, edges, or graphs, GNNs can accomplish tasks like predicting categories of nodes or even graphs without labels, i.e. node classification [21] and graph classification [22], and predicting missing links between nodes, i.e. link prediction [23]. The models of GNNs are in the early stages, but we have seen rapid growth. Graph Convolution Networks (GCNs) [21], [24] , GraphSage [25], and Graph Attention Networks (GATs) [26] are several novel GNN architectures proposed in recent years.

GNNs combine graph propagation and neural network operations [27], leading to the complexity of the GNN training process. Each target node needs to aggregate its neighborhood node or edge features with its own original feature to generate the new feature. Then, each target node will produce one computing sub-graph, which is determined by the graph structure. Thus, the training of GNNs involves a sparse memory access pattern. The neural networks are applied to transform these collected features. And the more layers of GNNs are used, the more nodes will be required when computing one target node. So, it is often too costly to take all of the neighbors of the target nodes. The neighbor sampling strategy which selects a subset of neighbors is proposed from GraphSage [25] to alleviate the problem and enable inductive learning. This reduces the required neighbor nodes during the mini-batch stochastic GNN training process. Considering the above, the main difference between training GNNs and Deep Neural Networks (DNNs) is the additional information aggregation of neighbors which leads to sparse computing patterns. Actually, Wang et al. [28] have proven that the use of general sparse matrix computing is one of the most time-consuming parts of training GNNs.

With the prevalence of graph-structured data in real-world applications, large-scale graphs with billions of edges or a dataset with millions of graphs are created to enable better results for different realistic tasks. However, the efforts to extend graph neural networks to process a large-scale graph using

SC22, November 13-18, 2022, Dallas, Texas, USA
978-1-6654-5444-5/22/$31.00 ©2022 IEEE

distributed training are extremely limited. The problem tangles the distributed graph operations related to graph storage with the distributed neural network training, making it more complex than the traditional dense distributed neural network training problems. There are several available frameworks for distributed GNN training. Most of them store graph structures and features on CPU and train neural networks on GPU, such as the Deep Graph Library (DGL) [28] and Pytorch Geometry (PyG) [29], of which the communication between CPUs and GPUs often bounds the training performance.

Considering the above, since the combination of graph propagation and neural network operations for GNNs, there are several challenges to distributed training GNNs on large-scale graphs efficiently. Firstly, as each target node has one sub-graph computation, combined with their node features, the memory required will become significantly large, even with a small batch size. These nodes and features will be transferred from CPU to GPU, leading the network communication to be a bottleneck for GNN training. Secondly, the memory access of neighborhood node features of one target node is sparse. Either collecting sparse features on CPU before sending them to GPU or directly accessing these sparse features of CPU from GPU leads to high pressure on PCIe. Furthermore, the neighbor sampling for decreasing neighborhood nodes and other sparse computing operations aggravates the problem. Due to the above reasons, GPUs are mostly waiting for data during training, thus their utilization is very low.

In this paper, we propose WholeGraph, a fast GNN training framework for a multi-GPU distributed shared memory system on large-scale graphs. We treat the device memory of all GPUs in the system as a whole to solve the limited GPU memory problem. The graph structure and node or edge features are stored across different GPUs. In this way, the communication bottleneck between CPU and GPUs is eliminated. Using GPUDirect Peer-to-Peer (P2P) memory access and CUDA IPC techniques, we can implement different GPU memory accesses using different processes in one CUDA Kernel. Thus, gathering neighbor node or edge features from other GPUs can be implemented using one CUDA kernel. Furthermore, we optimize several ops used in GNN training, such as multi-GPU sampling and other GNN layer ops. We build WholeGraph on top of the popular deep neural network training framework Pytorch [30]. Without loss of generality, WholeGraph with our optimized ops also can be used in inference scenarios, since it does not require collective communication. And considering the multi-GPU platform as a distributed shared memory architecture is also appropriate for other sparse graph computing patterns.

To summarize, our main contributions are:

- We propose a novel GNN training framework with multi-GPU distributed shared memory architecture, in which the graph structure and node or edge features are stored across multi-GPUs. In this way, the communication between CPU and GPU can be eliminated.
- Based on graph structure and node or edge features storage across multi-GPUs, we implemented optimized multi-GPU sampling and other optimized ops including sparse matrix computing and multi-GPU gathering features to further improve communication efficiency.
- The experiments show that the speedups of WholeGraph can be up to 57.32X and 242.98X compared to DGL [28] and PyG [29], respectively. And the GPU utilization ratio of WholeGraph can maintain 95% or above during GNN training process. Also, our experiments show that the multi-node scaling is almost linear.

II. Background

In this section, we will have a brief introduction to GNNs, and then discuss the multi-GPU peer-to-peer access technique.

A. Graph Neural Networks

GNNs are one class of neural networks operating on graph-structure data. Most of GNNs can be expressed as message passing scheme [31], which can be denoted as follows:

$$x_t^l = \Gamma^l(x_t^{l-1}, \oplus_{s\in\mathcal{N}(t)}(\varphi^l(x_t^{l-1}, x_s^{l-1}, e_{s,t}^{l-1}))) \quad (1)$$

where x_t^l is the learned embedding of target node t at layer l, and x_s^{l-1} denotes the embedding of node s, which is the neighbor node of target node t, at layer $l-1$. $e_{s,t}^{l-1}$ represents the edge embedding from node s to node t. $\mathcal{N}(t)$ denotes the set of neighbors of target node t. φ calls the message function, $\oplus$ calls the aggregation function, and Γ is the update function. There are several popular GNN models, like GCN [21], [24], GraphSage [25], and GAT [26], applicable to formula (1). The differences between these GNN models involve different message, aggregation, and update functions.

When we train all of the nodes in one graph simultaneously, it is called full batch training. However, the full batch training has some drawbacks. At first, it consumes memory a lot, as it needs to store all activation of every layer for all nodes. When the graph is large, the required memory may exceed the system's memory capacity. Secondly, the parameter is updated only once for one epoch training. This slows down the convergence severely. In view of the above shortcomings, we usually use mini-batch training for GNNs.

When the graph is significantly large, the sampling techniques [25] have been proposed to reduce the number of neighbor nodes required to aggregate. Figure 1 shows the current workflow of sampling based mini-batch training for GNNs. The input graph and its corresponding node or edge features are stored in host memory. Each training iteration mainly consists of four steps, namely sampling, gathering feature, GPU data loading and training on GPU. In every training iteration, the sampler will randomly choose one batch size of training nodes as the training target nodes. Then the sampler traverses the graph structure to collect some of the neighbor nodes of the target nodes layer by layer, generating the computation sub-graph for these target nodes. According to these sampled nodes, the corresponding features are gathered to form the mini-batch features. These two steps are completed on the CPU. Then the prepared training data including mini-batch graph and mini-batch features are transferred to GPU

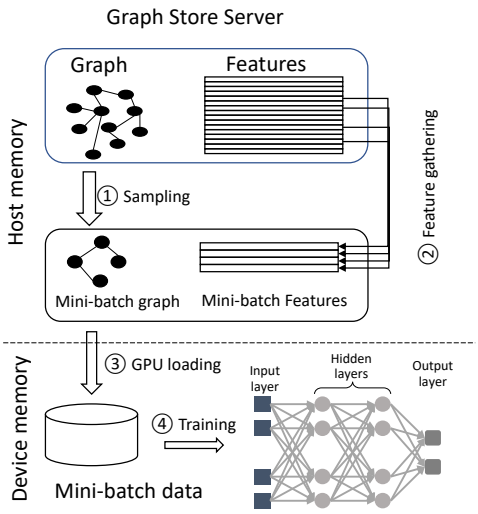

Fig. 1. The workflow overview of existing GNN training.

from CPU. This will be one of the bottlenecks during the whole training process, as with neighbor sampling, the feature size can be very large. The final step is training GNNs on GPU. To get converged accuracy, we need to train the model with lots of epochs, and one epoch includes many iteration steps.

B. Distributed Shared Memory System

The distributed memory system is a multiprocessor or multi-node computer system, in which each processor or node can only directly access its own private memory. The data in remote processors or nodes must be transferred through the interconnection between processors or nodes using message passing. It is usually the programmer's task to define how and when data is communicated. Most computers adopt the distributed memory system since it's cost-effective and easy to be implemented, leaving the communication task to the programmer. On the contrary, the shared memory system offers a global address space used by all processors or nodes. In the shared memory system, all processors can access the whole physical memory no matter where the data resides. It provides a user-friendly programming perspective. Distributed shared memory system is a memory architecture where the separated physical memory can be treated as one logically shared address space. In this system, each processor or node of a cluster can access a large shared memory in addition to its own limited private memory. So it is not only user-friendly but also more easily scales with the algorithm.

On a multi-GPU system, the most common way to use memory is to treat multiple GPUs as a distributed memory system. Communication between multi-GPUs is handled by programmers with either cudaMemcpy, NCCL library, or MPI. However, there is also the possibility to use multi-GPU systems such as DGX-A100 as a distributed shared memory system. NVIDIA provides two mechanisms for a GPU to access other GPU's memory. One is the unified memory (UM) technique, the other is based on GPUDirect Peer-to-Peer (P2P) Memory Access. UM is implemented based on software interrupt and page migration. That means the granularity is one page. When a GPU accesses one memory address that is not in its own local memory, a page fault is triggered. CPU will handle the page fault, perform the page migration, and set up a new page table for that GPU. Then the GPU's memory access can be completed. This long process will lead to high latency. In contrast, users can use *cudaDeviceEnablePeerAccess()* to enable peer access between GPUs. After that, GPU0 can directly access GPU1's memory (load/store) within the CUDA kernel code, i.e., GPUDirect P2P memory access. In this mechanism, memory access is handled by hardware without OS kernel or driver involvement. So, the latency is lower.

Memory Size (GB)	Latency(us)	
	UM	Peer Access
8	20.8	1.35
16	29.6	1.37
32	32.5	1.43
64	35.3	1.51
128	35.8	1.56

TABLE I
THE UM AND GPUDIRECT P2P MEMORY ACCESS LATENCY

Table I shows the detailed experimental data about UM and GPUDirect P2P memory access latency. In the experiment, we allocate one large memory, randomly initialized. Then, one thread accesses the memory in a specific way. That is, according to the value just visited and the step id (which adds 1 after each memory access), the thread determines the next memory access address which is across the large memory space allocated. Thus, through the memory access dependency chain, the access latency cannot be hidden by software and hardware. The number of memory access is 100K. And we calculate the time of one step of memory access which is the access latency. For UM, the memory is allocated using *cudaMallocManaged()*, and each GPU randomly initializes one equal-sized partition of the memory, guaranteeing allocated memory distributed across multi-GPUs. For P2P memory access, the memory is allocated equally across all GPUs using *cudaMalloc()*. The latency of UM and GPUDirect P2P memory access is shown in Table I. As we can see, the average latency for UM is 20 to more than 30 microseconds, while GPUDirect P2P is at the order of 1-microsecond magnitude.

In this work, we treat the multi-GPU system as a distributed shared memory system implemented using GPUDirect P2P memory access technology. WholeGraph is built on this mechanism. The experiments in the evaluation section also prove that the performance of our GNN framework implemented based on a multi-GPU distributed shared memory system is better than that of a multi-GPU distributed memory system.

III. OUR METHODS

A. Overview

WholeGraph is a GNN training framework built on multi-GPU distributed shared memory architecture, shown in Figure 2. To achieve high-performance memory access across

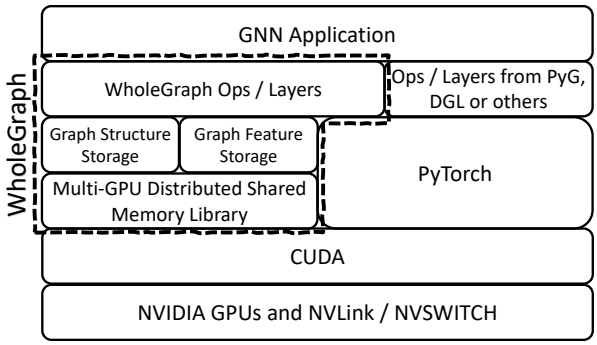

Fig. 2. The software stack of WholeGraph.

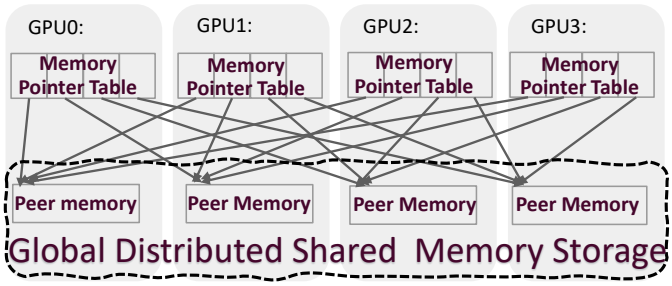

Fig. 3. Multi-GPU distributed shared memory library implementation.

GPUs, we construct a multi-GPU distributed shared memory library based on GPUDirect Peer-to-Peer memory access technology. The library manages all of the memory of multi-GPUs, providing unique host and device handles for upper layers of the software stack of WholeGraph. Then leveraging the library, we are able to store graph structure and graph features to the whole memory of multi-GPUs, significantly larger than that of a single GPU. For GNN training, we make use of the automatic differentiation module in PyTorch [30] which is one of the most prevalent machine learning frameworks. To use multi-GPU distributed shared memory, we implement two GNN ops including GPU sampling and multi-GPU gathering feature ops. Based on our multi-GPU distributed shared memory library, GPU sampling and multi-GPU gathering feature ops, users have two ways to implement their own GNN applications using our WholeGraph. One is through GNN layers from third-party GNN frameworks such as PyG [29] and DGL [28]. The other is using our optimized GNN layers, which is more efficient than that of PyG [29] and DGL [28]. The performance of GNN layers can be seen in Section IV-C5.

B. Multi-GPU Graph Storage

The multi-GPU distributed shared memory library is implemented as shown in Figure 3. Each GPU holds part of the data which is stored in peer memory. For GNN training, one process usually controls one single GPU. However, any device memory pointer or event handle created by a host thread cannot be directly referenced by other threads belonging to a different process. To make a process belonging to one GPU accesses data in another process mastered by a peer GPU, we use GPU inter-process communication (IPC) technology provided in CUDA, i.e. CUDA IPC. CUDA IPC is to address data movement overheads between processes belonging to different GPUs connected to the same machine node.

To set up the shared memory library, each process first allocates the part of memory that lies in its GPU and exposes the device memory pointer by CUDA IPC handle using *cudaIpcGetMemHandle()* API. After all processes have obtained their own IPC handle, an *AllGather* pattern communication is performed so that each process can get all the CUDA IPC handles from other processes. Then *cudaIpcOpenMemHandle()* API is used to convert the IPC handle from the other process to a local device pointer. In this way, the device memory belonging to one process can be referenced by all others. Then a memory pointer table is allocated on each device, shown in Figure 3. It is used to store mapped pointers of peer memory from each GPU. This allows each GPU to directly access others' device memory within a CUDA kernel. The Memory Pointer Table is just an array of pointers, one for each GPU. For DGX-A100 with 8 GPUs, it is just $8 \times 8 = 64$ bytes. So this will not hurt scalability. After all this, all the memory allocated can be accessed by all processes. As this procedure needs *cudaMalloc()* and inter-process communication, it is likely tens to one or two hundred of milliseconds required to set up one piece of distributed shared memory, depending on the memory size. However, this procedure is only at the beginning of the whole GNN training process. After the graph structure and features are constructed, the setup of multi-GPU shared memory is completed. During the training iterations, there is no need to do this setup.

In the meantime, leveraging the library, high bandwidth is also guaranteed by the NVLink connections between GPUs. The theoretical unidirectional bandwidth of NVLink is 300 GB/s on a DGX-A100 system. If the graph and feature of GNN are stored in host memory, the transmission between host and GPU goes through PCIe. The maximum bandwidth of PCIe 4.0 x16 is 32 GB/s. However, in most servers, usually more than one GPU shares one PCIe uplink port, for example, in DGX-A100 system shown in Figure 6, two GPUs and two NICs share one PCIe host x16 port, when all the GPUs are communicating with host memory, each GPU can get only one half of the PCIe 4.0 x16 bandwidth, namely 16 GB/s. If we just consider the bandwidth of data transfer, WholeGraph has a theoretical speedup of 18.75X, compared to any GNN training system based on host memory storage.

Having this high bandwidth multi-GPU distributed shared memory, we then implement the graph structure storage and graph feature storage. We partition the nodes of the graph to different GPUs according to the node ID hash value. Each graph node is assigned to a GlobalID, which is composed of rank ID and local ID. All the edges are stored together with the source node. Node features are also stored in the same GPU as the node. Then, all graph-related data including structure and features are stored across multi-GPUs and accessible from each GPU.

C. WholeGraph Ops

In this section, we will present WholeGraph ops for GNN training, including sampling, gathering features, append unique, and some GNN layer ops. As mentioned in Section II-A, for a large graph, it is impossible to train on the entire graph, so the sampling-based mini-batch GNN training method is adopted. In this method, the neighbors of mini-batch target nodes are randomly sampled to construct the computing sub-graph using the sampling op. As identical nodes may be sampled by different target nodes, a unique set of the sampled nodes needs to be created using the unique op to remove the duplicate nodes. After that, the node features need to be gathered from the device memory of local or remote GPUs. Finally, the prepared data will be fed into the GNN layers to proceed forward and backward training.

1) Multi-GPU Sampling Op: Random sampling might also be the bottleneck of GNN training. Usually, GNN training systems need random neighbor sampling without replacement so that there are no duplicated neighbors for one target node. However, generating non-duplicate random numbers is not easy to parallelize as each thread has to know neighbors sampled by other threads. We implement a fully parallel GPU random sampling without replacement using the path doubling method [32] as shown in Algorithm 1. We denote the maximum sample count of each target node as M, and the neighbor count of the target node as N. The input of Algorithm 1 is M and N, and the output of it is M neighbor node indices stored in an array $res[M]$. There are some auxiliary arrays to help us with parallel sampling, like array $r[M]$, $chain[M]$, $s[M]$, $p[M]$, $q[M]$ and $last[M]$. At first it uses $random()$ function to generate M random number ranging from 0 to $N-1-i$ shown in line 2, and initializes array $chain[M]$ with the index of each element. Then, it sorts the array $r[M]$ using $parallel\_sort()$ function shown in line 5, and returns back the sorted array stored in $s[M]$ and the original index of each element $p[M]$. Line 6 to line 11 conditionally update the elements in array $chain[M]$. After that, array $chain[M]$ updates again using the path doubling function shown in line 12, i.e. $chain[i] = chain[chain[i]]$. Then, array $last[i]$ is updated using array $chain[M]$. Finally, using arrays above, we can get the result of M neighbor node indices shown in line 16 to line 22. From Algorithm 1, the random sampling without replacement can be performed in parallel.

When we implement the Multi-GPU sampling op, M threads in the thread block of CUDA are grouped together to generate the sampled neighbors for one target node. For the case M is no less than N, all of N neighbors are sampled, and each thread can simply output its id. For the case M is smaller than N, we need to randomly select M nodes from all of N neighbors using Algorithm 1. A few optimization implementations are made on GPU. For example, the function of $parallel\_sort()$ in line 5 of Algorithm 1 needs to return two arrays. One is for the sorted array, and the other is for the original index. We pack 32-bit array $r[M]$ and its index array to one 64-bit array, high 32-bit of which stores array $r[M]$ and low 32-bit stores the index. Then we use radix-sort method to sort the new 64-bit array. Thus we can easily get the results of function $parallel\_sort()$. Furthermore, we combine this algorithm with our Multi-GPU graph structure storage to create a very efficient multi-GPU sampling implementation.

Algorithm 1 Parallel random sample algorithm without replacement

Input: Maximum sample count M , Neighbor count N for each target node
Output: sampled neighbor array $res[M]$

1: **for** $i \leftarrow 0$ to $M-1$ **parallel do**
2: $r[i] \leftarrow random(N-1-i)$
3: $chain[i] \leftarrow i$
4: **end for**
5: $s, p \leftarrow parallel\_sort(r)$
6: **for** $i \leftarrow 0$ to $M-1$ **parallel do**
7: $q[p[i]] \leftarrow i$
8: **if** ($i = M-1$ or $s[i] \neq s[i+1]$) and $s[i] \geq N-M$ **then**
9: $chain[N-s[i]-1] \leftarrow p[i]$
10: **end if**
11: **end for**
12: $chain \leftarrow path\_doubling(chain)$
13: **for** $i \leftarrow 0$ to $M-1$ **parallel do**
14: $last[i] \leftarrow N - chain[i] - 1$
15: **end for**
16: **for** $i \leftarrow 0$ to $M-1$ **parallel do**
17: **if** $i = 0$ or $q[i] = 0$ or $s[q[i]] \neq s[q[i]-1]$ **then**
18: $res[i] \leftarrow r[i]$
19: **else**
20: $res[i] \leftarrow last[p[q[i]-1]]$
21: **end if**
22: **end for**

2) Append Unique Op: During neighbor sampling, the same nodes may be sampled from different target nodes. Thus, there might be a lot of duplicate nodes. To decrease the amount of gathering features from other GPU, it is better to get rid of these duplicate nodes. So we need to create a unique set of all the nodes. Further, for better reuse of the gathered feature, it is preferred to put all the target nodes in front. At the same time, we need to maintain the original sequence of these target nodes. To meet these requirements and do this efficiently, we implement an Append Unique op, which combines the operations of appending neighbor nodes to target nodes with removing duplicate nodes called unique op. In order to efficiently remove duplicate nodes, we adopt the hash table method instead of the sort method used in other frameworks. That is, we leverage the hash table to help us achieve appending the neighbor nodes to target nodes and removing duplicate nodes. The key of our GPU hash table is nodeID. The value of the hash table needs to store the ID in the sub-graph which is not known for the neighbor node at first. So, we assign the value of the hash table of the neighbor

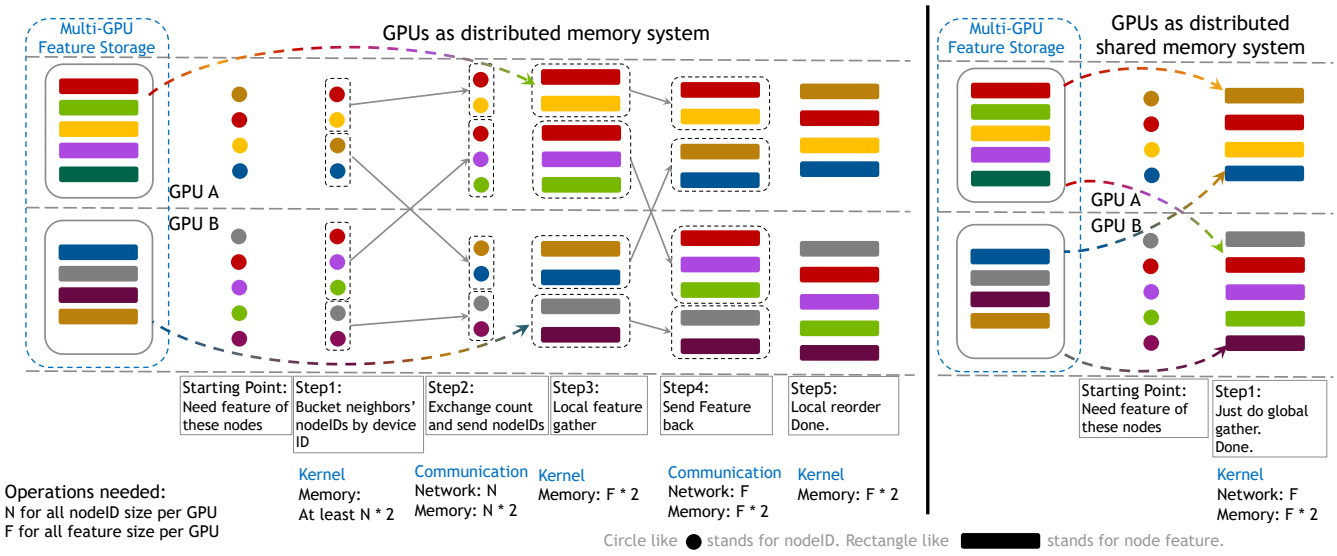

Fig. 4. Global gather in distributed memory system and shared memory system

node for -1 in the beginning. The value of hash table of target node is their indices in the target list, e.g. T0 ~T3 with their nodeIDs ID0 ~ID3 as shown in Figure 5. T0 ~T3 is the list of target nodes, and their indices are 0 ~3 in the list. The first key-value hash table in Figure 5 is the insertion process of target node. The second key-value hash table in Figure 5 is the insertion process of the neighbor node. When we insert the duplicate nodes into the hash table, we can easily know that the node has been inserted into the hash table. We do not need to insert the duplicate node into the hash table anymore. Now we achieve the goal of removing duplicate nodes. We use some ideas of inserting key-value into a hash table in Warpcore [33]. Warpcore is a library for hash table data structures on GPUs, with a parallel hashing scheme tailored to improve global memory access patterns. For the goal of appending neighbor nodes to target nodes, we need some extra effort. We need to assign the value of the hash table of unique neighbor nodes to ID in the sub-graph. To do this in a parallel manner, we separate the hash table slots into many buckets. We count the number of -1 in the value of the hash table and store the number in the bucket table as shown in the third pair table in Figure 5. Then we perform an exclusive prefix sum operation for the data in the bucket table, helping to get the start ID of neighbor nodes for each bucket. The number in each bucket after the exclusive prefix sum operation, add the total number of target nodes together, getting the final ID of neighbor nodes in the sub-graph shown in the last pair table in Figure 5. We can retrieve the ID in the sub-graph from the hash table for each node, including target nodes and neighbor nodes. The sparse adjacency matrix of sub-graph stored with CSR format in WholeGraph can be built based on the sampled edges and these assigned continuous IDs.

With multi-GPU sampling and append unique Ops, single layer sub-graph sampling can be done on GPUs. Multi-layer sub-graph sampling can be done by simply stacking multiple single-layer sub-graph sampling.

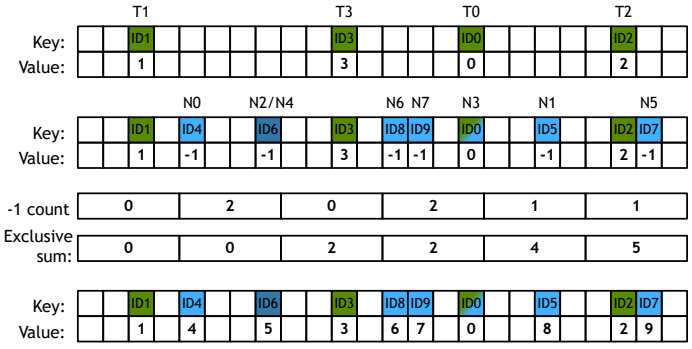

Fig. 5. GPU AppendUniqueOp implementation

3) Multi-GPU Gathering Feature Op: Another very common and time-consuming task in GNN training system is to gather random node features(e.g. neighbors' features of some target nodes) into the memory of the current device. The input of this task for each GPU device is a random list of node IDs, whose features may be in any of the GPU devices. The expected output is to gather the feature vector of the corresponding node and put them according to the order of the input list. In WholeGraph we call it global gather. As GPU can be used as both distributed memory system and shared memory system, here we analyze the implementation difference between these two kinds of systems.

In a distributed memory system, global gathering features can be accomplished in the following steps, as shown on the left of Figure 4. In this setup, each GPU cannot access others' device memory to directly fetch the node IDs. Therefore, this can be only accomplished by explicit communication like *ncclSend/ncclRecv*. Furthermore, to minimize the overhead of initiating communication and make better use of bandwidth,

node IDs to the same GPU need to be put next to each other and sent together. By doing this only one *Send*/*Recv* for each GPU pair is needed and the communication is more efficient. That is why bucket node IDs in step1 in Figure 4 are needed. After the node IDs are put in the bucket by their home GPUs, they are ready to be sent to their home GPU. But before their home GPU can receive that, the node ID count needs to be sent to their home GPU first, and then the real node IDs, as shown in step 2. After each GPU get all the node IDs, they can perform local gathering for all the GPUs shown in step 3. After that, in step 4 all the gathered features are sent back. To make them in the order of the input node ID list, a local reorder is needed as shown in step 5. And the operations on memory and network are also listed in Figure 4. As we can see, when using GPUs as a distributed memory system, there is quite a lot of communication, as well as more processing steps. This leads to both more network traffic and more use in memory bandwidth.

While in the shared memory system, as each GPU can directly access the memory of other GPUs', this can be done by a single gather kernel in each GPU. The underlying NVLink and NVSwitch will handle all the necessary communication without the involvement of software. As shown in the right of Figure 4, only one step and a minimum of operations are needed, with less network traffic and less use in memory bandwidth. Later experiments show the performance difference between these two implementations.

4) GNN Layer Ops: With the sub-graph sampling Ops and multi-GPU gathering feature Op, the sub-graph structure and its corresponding features are all in the training GPU, making it possible to do later work on this single GPU. Then it is possible to use GNN layers from DGL [28] or PyG [29] to build various GNN models. In addition, to make the GNN models run faster, we have also implemented some high-performance GNN layer Ops.

As the sub-graph is represented by a sparse adjacency matrix, GNN-related Ops like message passing in Eq. 1 need to work on these sparse matrices. We implemented some sparse matrix operations like generalized sparse matrix dense matrix multiplication (g-SpMM) and generalized sampled-dense-dense matrix multiplication (g-SDDMM). In WholeGraph, we store the adjacency matrix in CSR format, with a duplicate count for each sub-graph node indicating how many times the node is sampled as a neighbor node. This duplicate count is generated as an output of our Append Unique Op. For message passing, it is a g-SpMM pattern as the message passes from edges to the target node and aggregates in the target node. One of the two inputs is the sub-graph node feature which is a dense matrix. The other is the sparse adjacency matrix with optional edge weights. The forward of g-Spmm can be done directly on the CSR matrix. Backward edge weights can be done by a g-SDDMM also on the CSR matrix. Backward dense feature input should be g-SpMM on the transposed CSR matrix, this can be done by computing on the original CSR matrix and using atomic add operations to avoid the sparse matrix transpose. Since for large graphs, it is likely that there are fewer duplicated samples. We use the duplicate count array to help identify the nodes without duplicated one, whose atomic add can then be optimized to a simple assign operation to get better performance.

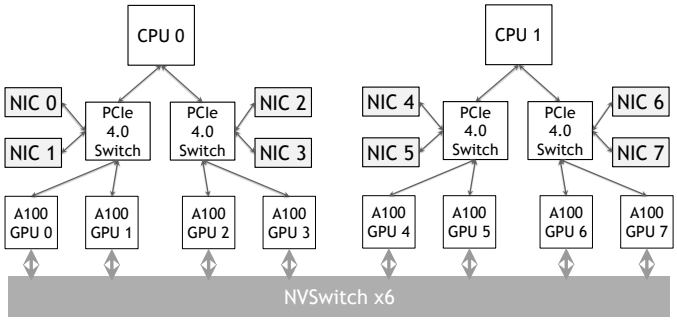

Fig. 6. DGX-A100 system topology

D. Multi-Node Support

To further speed up GNN training with more machines, WholeGraph supports multi-node training with the data parallel approach, using the Apex DistributedDataParallel package [34]. Each machine node holds one replica of the graph structure and graph features, using a multi-GPU distributed shared memory library. At each training step, each GPU performs the training with its own training samples. Sampling and gathering feature ops are proceeded using graph and feature stored in local machine node. The forward and backward computing stage is performed on a single GPU. Afterward, all GPUs synchronize the computed gradients with each other using the Allreduce communication, before updating the model parameters. Finally, each GPU has the same GNN model parameters.

IV. Evaluation

Experimental Setup: Our experiments are conducted on a DGX-A100 system with 8 NVIDIA A100 GPUs. Figure 6 shows one DGX-A100 topology. There are two AMD 64-core Rome 7742 CPUs in this system, consisting of a total of 128 cores. Under one PCIe 4.0 switch, there are 2 GPUs and 2 InfiniBand ConnectX-6 NICs. All of 8 NVIDIA A100 GPUs are connected to NVSwitch with 600 GB/s NVLink bi-direction bandwidth or unidirectional bandwidth of 300 GB/s. The version of CUDA used in the experiments is 11.4.

DataSets: The datasets we used in our evaluation are listed in Table II. Ogbn-products dataset is a medium size graph, recording an Amazon product co-purchasing network from OGB repository [35]. Ogbn-papers100M is a large citation graph with 111 million papers indexed. ogbn-products is larger than any existing node classification datasets from OGB repository [35]. The node features of ogbn-product and ogbn-papers100M are constructed by OGB repository [35]. The task of these two datasets is to predict the category of a node in a multi-class classification setup. Because these two datasets have node features and labels, we use them to prove the

Graph	Nodes	Edges	Features
ogbn-products [35]	2.4M	61.9M	100
ogbn-papers100M [35]	111.1M	1.6B	128
Friendster [36]	68.3M	2.6B	128
UK_domain [36]	105.2M	3.3B	128

TABLE II
GRAPH DATASET USED IN EVALUATING WHOLEGRAPH

correctness of WholeGraph. Friendster and UK_domain are two other large graphs from the Koblenz Network Collection [36]. As node features are not provided by the collection, we randomly generate them. The ratio of labels of them is 1%, making 80% of the label data to be trained data, 10% to be test data, and 10% to be validation data. So we use these two graphs just to verify the performance of WholeGraph.

GNN Models: We use three prevalent GNN models, i.e. GCN [21], [24], GraphSage [25], and GAT [26], to evaluate WholeGraph. The batch size for three GNN models is 512. The number of layers of these three GNN models is three, and the hidden size is 256 for each layer. The original GCN [21], [24] does not use the sampling method, so the large datasets are not supported. To solve the problem, We add the sampling strategy to GCN [21], [24] model. Each layer of all the models uses the same sampling count of 30. There are several aggregation types for GraphSage [25]. We use the mean aggregation to present the performance of GraphSage [25]. The numbering head of GAT [26] is 4.

Baselines: We compare WholeGraph with two popular open-source GNN frameworks, i.e. DGL [28] v0.7.2 and PyG [29] v2.0.2 . We observe that using DGL 0.7.2 with CUDA support installed using pip will result in a lot of expensive cudaMalloc/cudaFree API calls, which makes the training process slower. So we compile DGL from the source with the flag BUILD_TORCH=ON and set USE_TORCH_ALLOC=1 to avoid this behavior, making DGL to run faster. We demonstrate that the accuracy of WholeGraph is comparable to that of DGL [28] and PyG [29], and the performance of WholeGraph outperforms them.

A. Accuracy

Here, we evaluate the correctness of WholeGraph. Figure 7 shows the validation accuracy of DGL and WholeGraph on ogbn-products for training a GraphSage model. As we can see in Figure 7, WholeGraph and DGL almost achieve the same accuracy on ogbn-products dataset iteration by iteration. Table III shows the detailed validation and test accuracy of PyG [29], DGL [28] and WholeGraph on ogbn-products and ogbn-papers100M datasets after training about 24 epochs. Again, we observe that PyG, DGL and WholeGraph can achieve almost the same validation and test accuracy for the same epochs.

B. Memory usage

We measured the memory consumption of WholeGraph at different phases by adding a pause and monitoring the GPU memory using *nvidia-smi*. The result is shown in Table IV

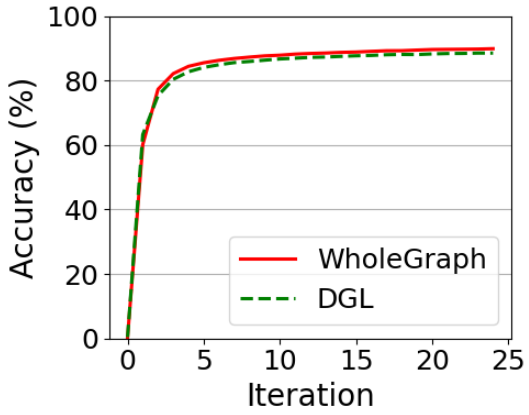

Fig. 7. The validation accuracy of DGL and WholeGraph on ogbn-products for training GraphSage model.

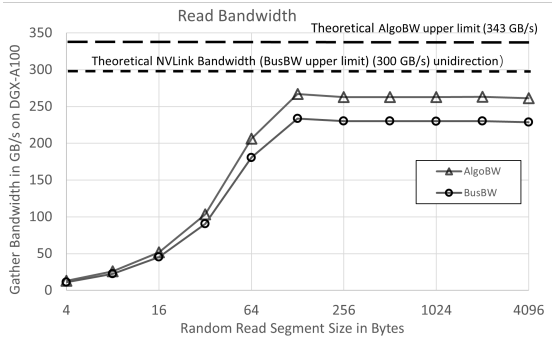

Fig. 8. The random memory access bandwidth of our multi-gpu distributed shared memory library according to different continuous segment size.

for ogbn-papers100M dataset. As shown in Table II, ogbn-papers100M has 1.6B edges. Since we use it as an undirected graph, the actual requirement to store it in memory is 3.2B edges. We use 8 bytes to store each edge, so the total memory required is 24 GB. That is 3 GB per GPU for a single node with eight GPUs. Our measured memory usage for graph structure is 3.1 GB shown in Table IV. The node feature dimension of ogbn-papers100M is 128, and the data type is float. So each node feature needs 512 bytes. As there are 111.1 million nodes for ogbn-papers100M, the memory of node feature required is 53 GB, with 6.6 GB per GPU. The measured memory usage of node feature per GPU is 6.7 GB. This proves that both the graph structure and node features are stored across all GPUs in the system. Besides, the memory usage of the training process per GPU is about 20.4 GB, involving model parameters, intermediate results, gradient and so on.

C. Performance

1) Multi-GPU Distributed Shared Memory Library: To evaluate the bandwidth of our Multi-GPU Distributed Shared Memory Library, we design an experiment. At first, we allocate a large memory distributed across all GPUs (128 GB) using our library. Each GPU concurrently gathers 4GB of

Graph	Model	DGL		PyG		WholeGraph	
		Valid	Test	Valid	Test	Valid	Test
ogbn-products	GCN	91.09%	78.02%	91.41%	76.86%	91.51%	78.46%
	GraphSage	91.30%	77.73%	92.33%	78.29%	92.02%	78.25%
	GAT	89.97%	77.55%	90.77%	78.72%	90.58%	78.16%
ogbn-papers100M	GCN	66.17%	63.73%	65.55%	63.19%	65.98%	63.41%
	GraphSage	68.28%	65.25%	68.28%	65.16%	68.14%	64.94%
	GAT	67.79%	64.71%	68.33%	65.10%	68.21%	65.21%

TABLE III
THE VALIDATION AND TEST ACCURACY OF PYG, DGL AND WHOLEGRAPH ON TWO REAL DATASETS.

	Measured Memory Consumption (GB) per GPU	Theoretical Total Memory (GB)
Graph Structure	3.1	24
Node Feature	6.7	53
Training	20.4	-

TABLE IV
THE MEMORY USAGE OF WHOLEGRAPH FOR OGBN-PAPERS100M ON A DGX-A100 SYSTEM WITH 8 NVIDIA A100 GPUS.

Dataset	GNN Model	Epoch Time PyG	Epoch Time DGL	Epoch Time Ours	Ours V.s. PyG	Ours V.s. DGL
ogbn-products	GCN	225.97	26.05	0.93	242.98	28.01
	GraphSage	228.96	30.8	0.99	231.27	31.11
	GAT	246.81	29.21	3.28	75.25	8.91
ogbn-papers100M	GCN	358.58	220.28	5.7	62.91	38.65
	GraphSage	314.88	273.67	6.0	52.48	45.61
	GAT	404.66	269.7	24.25	16.69	11.12
Friendster	GCN	286.78	159.48	2.79	102.79	57.16
	GraphSage	262.45	167.96	2.93	89.57	57.32
	GAT	287.76	154.56	12.83	22.43	12.05
UK_domain	GCN	122.61	77.1	2.77	44.26	27.83
	GraphSage	127.48	77.38	3.01	42.35	25.71
	GAT	153.71	85.1	10.85	14.17	7.84

TABLE V
THE AVERAGE EPOCH TIME AND SPEEDUPS OF TRAINING GCN, GRAPHSAGE AND GAT MODELS ON FOUR DATASETS.

data randomly distributed in all of the 8 GPUs. We have two ways to compute the bandwidth. One is the gathered datasize divided by time, denoted AlgoBW. It is the bandwidth seen by the algorithm. The other bandwidth is called BusBW, which is the bandwidth seen by the hardware "bus"(NVLink). Each GPU in DGX-A100 needs to communicate with 7 other GPUs over NVLink instead of 8. So BusBW equals to $\frac{7}{8}$ of AlgoBW. Here the maximum value of BusBW is 300 GB/s, i.e., the theoretical upper bandwidth of NVLink. The maximum value of AlgoBW is $300 \div (\frac{7}{8}) = 343$ GB/s. To demonstrate the ability to randomly read different sizes of continuous data, we change the continuous data read the size from 4 bytes to 4096 bytes, which we call random read segment size. The result is shown in Figure 8. As we can see, when the random read segment size is less than 64 bytes, the achieved bandwidth is almost proportional to the random read segment size. BusBW is about 181 GB/s, when the random read segment size gets to 64 bytes. And when the random read segment size is larger than 128 bytes, BusBW reaches and maintains around 230GB/s. AlgoBW is about 260GB/s. As we can see, with NVLink, a very small random read segment size (64 to 128 bytes) is needed to saturate NVLink's bandwidth. For training GNN models, the dimension of the node feature vector is generally at the level of a few hundreds, meaning that the random read segment size is hundreds to thousands of bytes. So our multi-GPU distributed shared memory library can perform well in gathering feature of training GNN. And the detailed performance results of the gathering feature are shown in Section IV-C4.

2) Overall Performance: We compare DGL, PyG and WholeGraph in terms of average epoch time for each dataset and GNN model, training on a single DGX-A100 system with 8 GPUs. Results are shown in Table V. We see that WholeGraph has about 1 or 2 orders of magnitude faster than DGL [28] or PyG [29] on all GNN models and datasets. The speedups shown in Table V range from 14.17x to 242.98x and 7.84x to 57.32x compared to PyG [29] and DGL [28], respectively. WholeGraph can achieve better performance on GCN and GraphSage models than that of GAT model. This is because GAT model has more parameters and computation amounts in the training stage. The ratio of computing in the whole training process (including sampling, gathering features, and computing) of GAT is higher than that of GCN and GraphSage. However, the bottleneck of PyG and DGL is sampling and gathering features, which can be seen in Section IV-C3. That is, the performance gains of WholeGraph are mainly from optimized sampling and gathering features. The different percentage of computing (i.e. training phase in Figure 9) leads to different speedup ratios.

3) Breakdown of Epoch Time: To help understand more about the performance results, we break down the training epoch time of GNN models on ogbn-products and ogbn-papers100m datasets into three parts. The first part is sampling, in which sub-graphs are generated and transferred to GPU. The second part is gathering features to GPU. For PyG [29] and DGL [28], the features are gathered and transferred to GPU from CPU memory storage. For WholeGraph, features are gathered by global gather feature Op from multi-GPU memory storage. The last part is training GNN models on GPU.

The results of the time breakdown are shown in Figure 9. We put different models of the same dataset and the same framework in one subplot as the epoch time between frameworks may vary a lot. We can see that for PyG [29] and DGL [28], the sampling and gathering features take most part of the time, especially for GCN and GraphSAGE models. The training part can hardly be seen from the figure. That is because the sampling and gathering feature part is the main

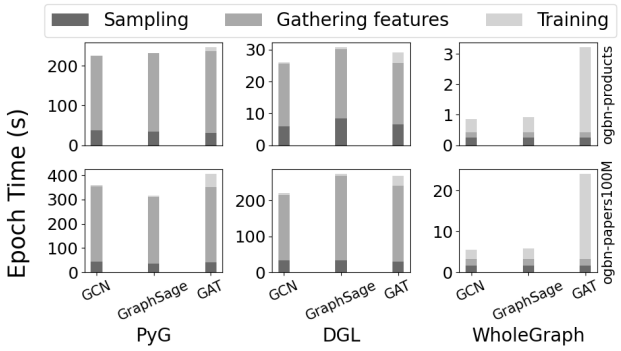

Fig. 9. The training epoch time breakdown of PyG, DGL and WholeGraph for training GCN, GraphSage and GAT models on ogbn-products and ogbn-papers100M datasets.

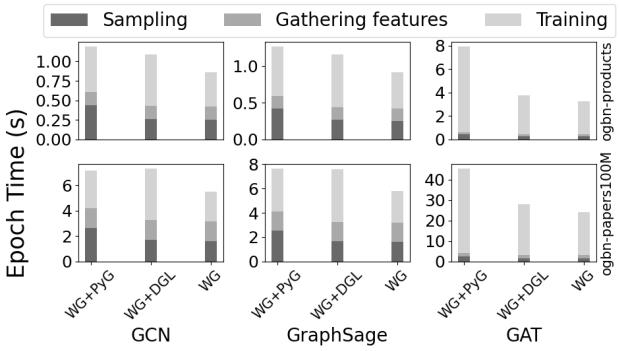

Fig. 11. The training epoch time breakdown of WholeGraph+PyG, WholeGraph+DGL and WholeGraph for training GCN, GraphSage and GAT models on ogbn-products and ogbn-papers100M datasets.

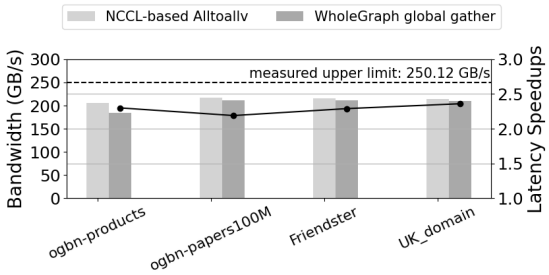

Fig. 10. The performance of gathering features. One is NCCL-based, and the other is ours. The latency speedups are the ratio of the whole gathering feature time of NCCL-based and ours. The bandwidth of NCCL-based gathring feature is just measured the final alltoallv. The bandwidth of ours is measured for the whole gathering featuring.

bottleneck for most PyG [29] and DGL [28]. For WholeGraph, we eliminated this bottleneck by our global distributed shared memory storage and optimized sampling and global gather Ops. We see that the time consumption of the sampling and gathering features part is much lower than that of the training part for WholeGraph.

4) Gathering Feature Performance: Our analysis in Section III-B and evaluation results in Section IV-C3 show that WholeGraph should and does have significant performance gain compared to that of graph storage using host memory like PyG [29] and DGL [28]. However, as we mentioned in Section III-C3, there are also different possible implementations using multi-GPU as graph storage. For example, gathering features from different GPUs of one machine node has two implementation methods. One is regarding the memory of different GPUs as distributed memory. The communication between different GPUs can be implemented using NCCL library. The other is ours, regarding the memory of different GPUs as a shared memory system. Here, we evaluate the latency and bandwidth of gathering features implemented by these two methods.

Figure 10 shows the experimental results. We test the total time of gathering feature for these two methods at each epoch during training real GNN models. The line chart in Figure 10 is the speedups of time for our method with that of NCCL-based gathering features, and the right y-axis shows the detailed speedup numbers. We can observe that the speedups of time are above 2X on all of datasets. The bar charts depict the bandwidth. For NCCL-based gathering feature method, we just compute the bandwidth of *Alltoallv()* process, i.e. step 4 shown in Figure 4. However, the time of NCCL-based gathering feature method consists of 4 steps shown in Figure 4. On the other hand, as the whole gathering features process of WholeGraph only has one kernel, the computing time of bandwidth is all of the running time of this kernel. Here all the bandwidth values are BusBW. As we can see in Figure 10, the bandwidth of these two is close to each other and all close to the measured NVLink upper limit, meaning that ours are well optimized, having comparability with *Alltoallv()*. And the overall time of WholeGraph's gathering features is almost the same as single stage 4 of NCCL-based gathering features shown in Figure 4. The time of all other stages in NCCL-based gathering features shown in Figure 4 need to be added up to obtain its overall time. It is obvious that the implementation of WholeGraph gathering features is the better one.

5) Layer Performance: As mentioned in Section III-A, WholeGraph can also use GNN layers from other GNN frameworks like PyG and DGL, making it easy to utilize various kinds of GNN layers from third-party libraries. So we also build GNN models using layers from DGL and PyG, while the sampling and gathering features is sampling and global gather Op of WholeGraph. The time breakdown is shown in Figure 11. Note here we put the same model of the same dataset in one subplot for comparison of different layer implementation.

We see the bottleneck of sampling and gathering features for PyG and DGL can be eliminated with the help of WholeGraph. And the GPU utilization can also reach up to 95% with PyG or DGL layers together with WholeGraph's sampling and global gather. Compared with Figure 9 in Section IV-C3, the time

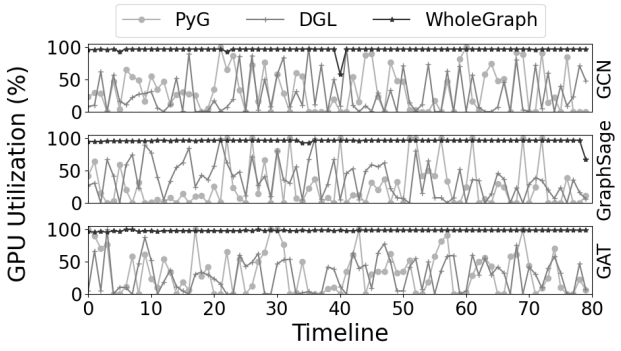

Fig. 12. The continuous GPU utilization ratio of training GNN models using PyG, DGL and WholeGraph on ogbn-papers100M dataset.

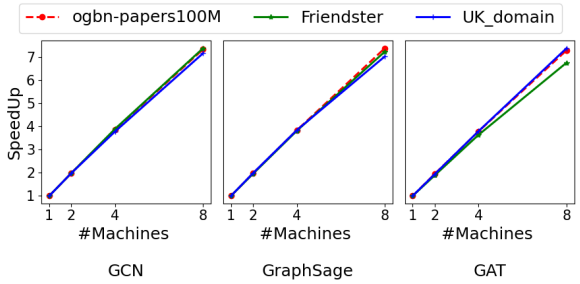

Fig. 13. The scalability of WholeGraph on three large datasets for GCN, GraphSage and GAT models.

of sampling and gathering features stages is reduced a lot by WholeGraph, and the time of the training stage stays almost the same as that of the original frameworks. We also see that with WholeGraph's optimized GNN layers, the overall epoch time can be speedup by up to 1.31x compared to WholeGraph using DGL layers and up to 2.43x compared to WholeGraph using PyG layers.

6) GPU Utilization: Figure 12 shows the GPU utilization ratio over time of one GPU in a single 8 GPUs server during training. As we can see, the GPU utilization ratio of PyG [29] and DGL [28] fluctuate significantly and fail to reach a high level, sometimes dropping to zero. This is because the data movement between CPU and GPU makes GPU wait for training data. This means that PyG [29] and DGL [28] are not able to leverage GPU resources efficiently. On the other hand, WholeGraph is able to keep GPUs busy during the GNN training process. And the GPU utilization ratio of WholeGraph can be maintained at a high level, 95% and above. There are two reasons for this. As we see from Figure 9, the bottleneck for DGL and PyG are sampling and gathering features, which limit the GPU utilization. WholeGraph's global distributed shared memory storage and high-performance global gather op eliminated the data transmission between CPU and GPU, which breaks the bottleneck of feature gathering. In addition, WholeGraph also provides high-performance GPU sampling based on global distributed shared memory storage which breaks the bottleneck of sampling. Thus WholeGraph breaks all the bottlenecks in multi-GPU GNN training and no wonder high GPU utilization can be achieved.

D. Multi-Node Performance

We evaluate the multi-nodes scalability of WholeGraph on three large datasets, i.e. ogbn-papers100M, Friendster, and UK_domain. Each node consists of 8 NVIDIA A100 GPUs. Figure 13 shows the speedup of average epoch time for training GCN, GraphSage, and GAT models, respectively. The training time on one node is regarded as the baseline and normalized to one. The results show that WholeGraph can achieve close to linear speedups on the three datasets for the three models with up to 8 nodes. To express the performance in another way, it means that we can train 80 epochs of a 3-layer GraphSAGE model with a hidden size of 256 and a sample count of 30,30,30 on the ogbn-papers100M dataset in 66 seconds, i.e. just over 1 minute, with 8 DGX-A100 servers.

V. Related Work

DGL [28] and PyG [29] are two popular GNN frameworks using a message passing interface. DGL [28] supports PyTorch [30], TensorFlow [37] and MXNet [38], while PyG [29] is built on PyTorch [30]. However, our experiments show that they fail to make the most of GPU resources due to low GPU utilization. AliGraph [39] is a GNN platform for CPU clusters, which does not support GPU training. It provides the CPU distributed graph storage. NeuGraph [40] supports multi-GPU GNN training, implemented on top of TensorFlow. As it stores intermediate GNN data in the host CPU DRAM, the bandwidth of the connection between CPU and GPU can become a bottleneck for the training performance. PaGraph [41] focuses itself on reducing the data movement between CPU and GPUs on a single node with multi-GPUs and is implemented on top of the DGL. ROC [42] is a distributed multi-GPU framework for GNN training on full graphs without using sampling techniques. So, it is not eligible for inductive representation learning on graphs and is limited by the graph size. P3 [43] proposes a distributed GNN training approach built on DGL. It combines data and model parallel and distributes the node features across different machine nodes. It communicates the activation at the first layer during the training process, which can reduce network communication under certain conditions. It is feasible to acquire performance improvement for some special cases, like GNN models with the small hidden size and large first layer sampled nodes, and fewer machine nodes. Seung et al. [44] provide a GPU-oriented data communication architecture for multi-GPU GNN training. Besides, NextDoor [45] was designed to perform graph sampling on a single GPU.

Nowadays, there are also some traditional graph libraries on GPUs, due to the high bandwidth of GPUs. The NVIDIA Graph Analytics library (nvGRAPH) provided in CUDA Toolkit, consists of parallel algorithms for high-performance

analytics on graphs. It supports PageRank, single source shortest path, and single source widest path. GunRock [46] is a graph-processing system running on GPUs. It supports traversal-based algorithms and ranking algorithms. However, WholeGraph is a graph neural network training framework with a multi-GPU distributed shared memory architecture. It combines graph operations and neural network training.

VI. Conclusion

GNNs are prevalent in recent years due to their capability to learn information from graph-structured data. Lots of GNN models have arisen. On the other side, the size of graphs is grown rapidly. The demand for processing large-scale graph datasets is getting stronger. In this work, we propose a novel and fast GNN training framework, i.e. WholeGraph, for multi-GPUs distributed shared memory system. As opposed to other GNN frameworks which store graph and node or edge features on CPU memory, we scatter these data on the memory of multiple GPUs, thus eliminating the communication between CPU and GPUs. Leveraging the GPUDirect P2P memory access technique, we can access sparse features stored on peer GPU memory from the current GPU. Furthermore, by optimizing some time-consuming ops, the performance is improved further. We implement WholeGraph on Pytorch and evaluate it using three GNN models and four realistic graph datasets. The experiments show that WholeGraph can achieve up to 57.32X and 242.98X speedup compared with DGL and PyG, respectively. The ratio of GPU utilization can sustain 95% during GNN model training.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[3] T. Young, D. Hazarika, S. Poria, and E. Cambria, "Recent trends in deep learning based natural language processing," *ieee Computational intelligenCe magazine*, vol. 13, no. 3, pp. 55–75, 2018.

[4] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal processing magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[5] M. Cho and D. Brand, "Mec: Memory-efficient convolution for deep neural network," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, p. 815–824.

[6] J. Liu, D. Yang, and J. Lai, "Optimizing winograd-based convolution with tensor cores," ser. ICPP 2021. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472473

[7] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2016, pp. 4013–4021. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.435

[8] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *ICML*, 2018.

[9] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, "Cnn-rnn: A unified framework for multi-label image classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2285–2294.

[10] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018.

[11] M. M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, and P. Vandergheynst, "Geometric deep learning: going beyond euclidean data," *IEEE Signal Processing Magazine*, vol. 34, no. 4, pp. 18–42, 2017.

[12] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[13] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2018, pp. 974–983.

[14] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The World Wide Web Conference*, 2019, pp. 417–426.

[15] N. Park, A. Kan, X. L. Dong, T. Zhao, and C. Faloutsos, "Estimating node importance in knowledge graphs using graph neural networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 596–606.

[16] D. Wang, J. Lin, P. Cui, Q. Jia, Z. Wang, Y. Fang, Q. Yu, J. Zhou, S. Yang, and Y. Qi, "A semi-supervised graph attentive network for financial fraud detection," in *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 2019, pp. 598–607.

[17] Y.-C. Lo, S. E. Rensi, W. Torng, and R. B. Altman, "Machine learning in chemoinformatics and drug discovery," *Drug discovery today*, vol. 23, no. 8, pp. 1538–1546, 2018.

[18] J. M. Stokes, K. Yang, K. Swanson, W. Jin, A. Cubillos-Ruiz, N. M. Donghia, C. R. MacNair, S. French, L. A. Carfrae, Z. Bloom-Ackermann *et al.*, "A deep learning approach to antibiotic discovery," *Cell*, vol. 180, no. 4, pp. 688–702, 2020.

[19] Y. Rong, Y. Bian, T. Xu, W. Xie, Y. Wei, W. Huang, and J. Huang, "Self-supervised graph transformer on large-scale molecular data," *arXiv preprint arXiv:2007.02835*, 2020.

[20] D. Yang, J. Liu, and J. Lai, "Edges: An efficient distributed graph embedding system on gpu clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 7, pp. 1892–1902, 2021.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[22] D. Bacciu, F. Errica, and A. Micheli, "Contextual graph Markov model: A deep and generative approach to graph processing," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 294–303. [Online]. Available: https://proceedings.mlr.press/v80/bacciu18a.html

[23] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in Neural Information Processing Systems*, vol. 31, pp. 5165–5175, 2018.

[24] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in neural information processing systems*, vol. 29, pp. 3844–3852, 2016.

[25] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[26] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," *International Conference on Learning Representations*, 2018, accepted as poster. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[27] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," 2021.

[28] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep

graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[29] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[31] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International conference on machine learning*. PMLR, 2017, pp. 1263–1272.

[32] V. Rajan, R. Ghosh, and P. Gupta, "An efficient parallel algorithm for random sampling," *Information processing letters*, vol. 30, no. 5, pp. 265–268, 1989.

[33] D. Jünger, R. Kobus, A. Müller, C. Hundt, K. Xu, W. Liu, and B. Schmidt, "Warpcore: A library for fast hash tables on gpus," in *27th IEEE International Conference on High Performance Computing, Data, and Analytics, HiPC 2020, Pune, India, December 16-19, 2020*. IEEE, 2020, pp. 11–20. [Online]. Available: https://doi.org/10.1109/HiPC50609.2020.00015

[34] NVIDIA, "Apex - a pytorch extension with nvidia-maintained utilities to streamline mixed precision and distributed training," 2021, last accessed 12 December 2021. [Online]. Available: https://github.com/NVIDIA/apex

[35] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.

[36] J. Kunegis, "KONECT – The Koblenz Network Collection," in *Proc. Int. Conf. on World Wide Web Companion*, 2013, pp. 1343–1350. [Online]. Available: http://dl.acm.org/citation.cfm?id=2488173

[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 2016, pp. 265–283.

[38] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[39] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2094–2105, Aug. 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352127

[40] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 443–458. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/ma

[41] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "Pagraph: Scaling gnn training on large graphs via computation-aware caching," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 401–415. [Online]. Available: https://doi.org/10.1145/3419111.3421281

[42] Z. Jia, S. Lin, M. Gao, M. Zaharia, and A. Aiken, "Improving the accuracy, scalability, and performance of graph neural networks with roc," in *Proceedings of Machine Learning and Systems*, I. Dhillon, D. Papailiopoulos, and V. Sze, Eds., vol. 2, 2020, pp. 187–198.

[43] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 551–568. [Online]. Available: https://www.usenix.org/conference/osdi21/presentation/gandhi

[44] S. W. Min, K. Wu, S. Huang, M. Hidayetoğlu, J. Xiong, E. Ebrahimi, D. Chen, and W.-m. Hwu, "Large graph convolutional network training with gpu-oriented data communication architecture," *arXiv preprint arXiv:2103.03330*, 2021.

[45] A. Jangda, S. Polisetty, A. Guha, and M. Serafini, "Accelerating graph sampling for graph machine learning using gpus," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 311–326. [Online]. Available: https://doi.org/10.1145/3447786.3456244

[46] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: Gpu graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, aug 2017. [Online]. Available: https://doi.org/10.1145/3108140

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

Our experiments are conducted on a DGX-A100 system with 8 NVIDIA A100 GPUs and the scaling experiment is on the NVIDIA Selene cluster with NVIDIA A100 GPUs. All 8 NVIDIA A100 GPUs are connected to NVSwitch with 600 GB/s NVLink bidirectional bandwidth or unidirectional bandwidth of 300 GB/s. The software versions are:

(1) Operating systems and versions: Ubuntu 20.04
(2) Compilers and versions: NVCC CUDA 11.4
(3) Libraries and versions: Python 3.8.10 and PyTorch 1.10.0a0+3fd9dcf
(4) Input datasets and versions: OGB ogbn-products and ogbn-papers100M

The GNN model parameters are:

(1) batchsize: 512
(2) num_layer: 3
(3) hiddensize: 256

Reproduction of the artifact without container: We ran our experiments on a DGXA100 system. We used the "nvcr.io/nvidia/pytorch:21.09-py3" docker image. Our repository has instructions on building our docker image according to our Dockerfile. To compile WholeGraph, from the source directory: mkdir build; cd build; cmake ../; make -j. There is a GNN example file to run our experiments, which includes our scripts.