DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs

2023-03-29
박현우

Table of Contents

1. Problem specification

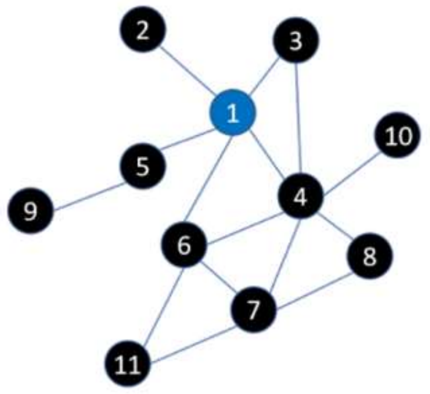
Hardships in distributed training GNN on giant Graph at scale:

1. Generate proper Mini-batch on large graph
2. Distributed parameters update on GNN
3. Overhead due to Network Communication in cluster.

1.1 Mini-batch Training on GNN

Mini-batch training on GNNs is difficult because:

1. Graph inherently represents the dependencies among training samples
2. each mini-batch must incorporate those depending samples whose number usually grows exponentially with Search depths (hops).



(a) An input graph.

Example: Mini-batch for Node 1 as Target

For search depth = 1 : [2, 3, 4, 5, 6]
For search depth = 2 : [], [4], [3, 6, 7, 8, 10], [9], [4, 7, 11]
…

1.2 Distributed Parameters update on GNN

1. Previous distributed GNNs are designed for Full graph computation expressed in the vertex-centric program paradigm, not edges.
   i.e. Entire clusters are synced to compute the full graph as a whole.

2. Existing domain-specific frameworks (DGL, PyTorch-Geometric) are mainly developed for single machine training.

1. Techniques in other domains can’t be well adopted to GNN due to vertex dependencies.

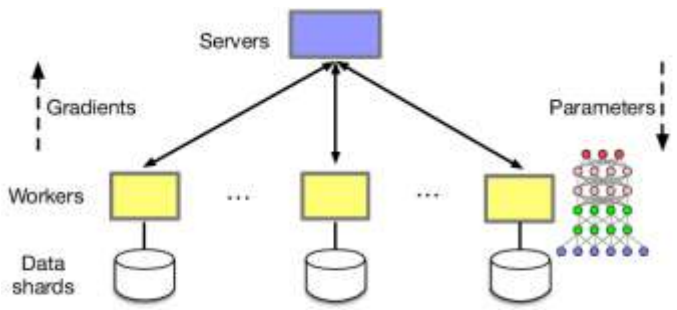
1. Others suffer from the huge network traffic caused by fetching neighbor node data.

1.3 Network Traffic Overload

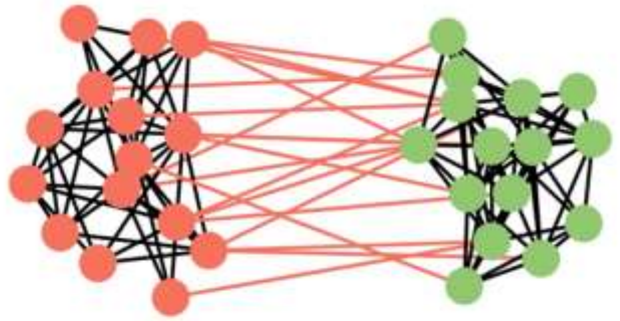
Due to the vertex dependency,
distributed GNN training requires to fetch hundreds of neighbor vertex data.

c.f. Exchanging the gradients on other domains.

Size of Data to transfer : Vertex Data >>> Gradients



▲ Distributed training on traditional DL.



▲ 2 mini-batches with lots of neighboring vertices

2.1. Contribution - Solve Hardships

1. To generate proper, well-balanced mini-batch:
   DistDGL allows ego-networks forming the mini-batches to efficiently include non-local nodes (i.e. Nodes in other partitions/machines).

1. For distributed training on GNN:
   DistDGL follows Synchronous training approach

1. To reduce network overhead:
   DistDGL adopts METIS, a high-quality and lightweight min-cut graph partitioning algorithm.

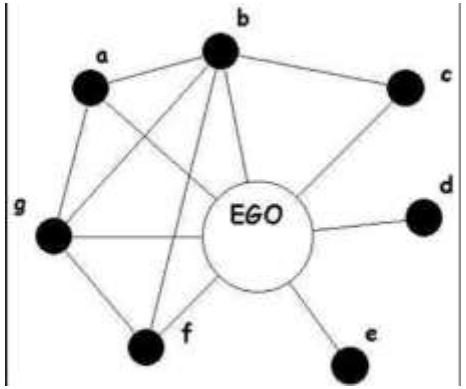
Terminology - Ego Network

Ego networks consist of

1. a focal node ("ego")
2. the nodes to whom ego is directly connected (these are called "alters")

In GNN context,

1. Ego node becomes target
2. Alters become features.



2.2. Contribution - Further achievements

1. Design Distributed training architecture for training GNN on giant graph.
2. Deploy multiple load balancing optimizations to further tackle partition imbalance issue.
3. Further reduce network communication in sampling.
4. Provide distributed embeddings with efficient sparse updates for transductive graph models.
5. Provide all these Distributed components with APIs compatible to DGL.

3. Distributed Training Architecture

DistDGL distributes the mini-batch training process of GNN models across a cluster of machines

Then, it trains the model by Synchronous stochastic gradient descent (SGD) training.

In procedure, each machine:

1. computes model gradients for its own mini-batch
2. synchronizes gradients with others
3. finally updates the local model replica.

Comparison against Async SGD
Pros:
1. No worry about Stale gradients.
2. Faster converge with larger step-size

Cons:
1. Bottleneck for each update
2. Poor robustness to machine failure.

3. Distributed Training Architecture

DistDGL consists of the following logical components:

1. Samplers
2. A KVStore
3. Trainers
4. A dense model update component

+ Graph Partitioning as Preprocessing before distributed training.

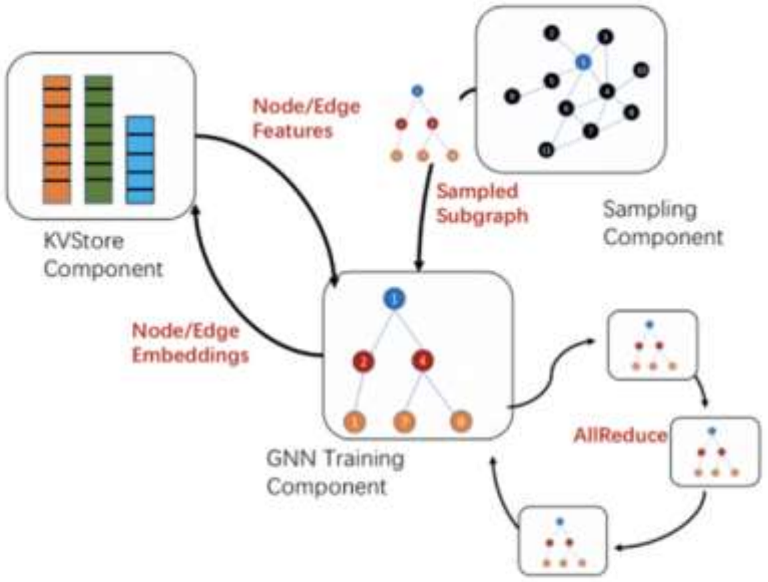


Fig. 2: DistDGL's logical components.

3.1 Graph Partitioning

Goal of Graph Partitioning:

1. To split the input graph to multiple partitions
   with a minimal number of edges (min-cut graph partitioning)
2. To balance partitions across cluster.

This leads effects of :

1. Reducing communication overheads among cluster.
2. Statically balance the computations.

DistDGL’s Graph Partitioning involves 2 steps:

1. METIS algorithm
2. Balancing partitions

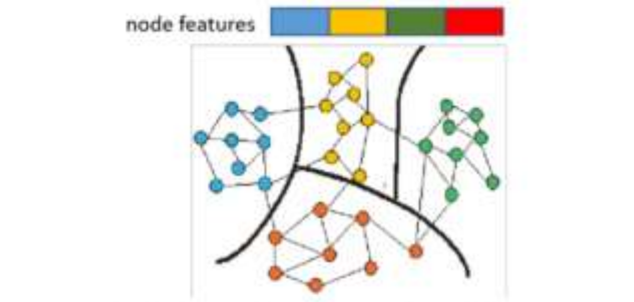
3.1.1 METIS

DistDGL adopts METIS as partitioning algorithm.

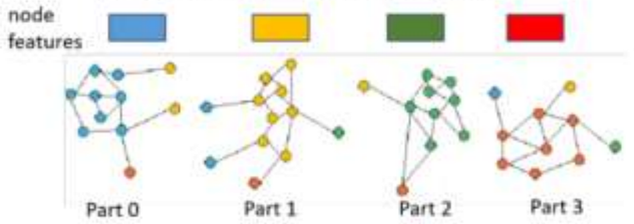
Procedure:

1. Assign densely connected vertices to the same partition to minimize the number of edge cuts between partitions.

2. Assign all incident edges to the same partition.



(a) Assign vertices to graph partitions



(b) Generate graph partitions with HALO vertices (the vertices with different colors from majority of the vertices in the partition).

Fig. 4: Graph partitioning with METIS in DistDGL.

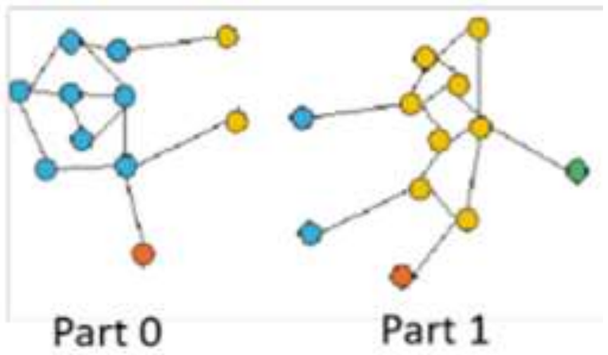
(2.) ensures that all the neighbors are accessible of the local vertices on the partition.

= Samplers can compute locally in most cases.

3.1.1 METIS

With METIS,

1. Each edge has a unique assignment
2. Some vertices may be duplicated.



For vertices in partition,

1. Core Vertices : Vertices uniquely assigned to a partition
2. HALO Vertices : Vertices duplicated by edge assignment strategy.

3.1.2 Balancing partitions

By default, METIS only roughly balances the number of vertices in a graph.

However, this is insufficient
to generate balanced partitions for synchronous minibatch training.

e.g. Partitions with densely connected vertices would have much more batches than those with sparsely connected.
(i.e. Num of batches increase faster as Search depth increases)

Sync minibatch training requires (Q)

1. The same number of batches from each partition per epoch
2. All batches to have roughly the same size

3.1.2 Balancing partitions

Allocate the same number of batches across all trainers per epoch

=> Can be formulated as Load Balancing Problem.

DistDGL leverages Multi-constraint mechanism in METIS
to solve such load balancing problem.

Multi-constraint mechanism is also used to balance

1. Training/Validation/Test set vertices/edges
2. vertices/edges of different types
   (~ Stratified sampling for imbalance dataset)

3.1.2 Balancing partitions

METIS algorithm is based on Multi-level paradigm.
i.e. Recursively partitioning by deepening search depths (hops).

To reduce complexity of both memory and computation,
DistDGL extends METIS to only retain a subset of the edges.

In addition, to transform coarser graphs into finer graphs in sampling process,
also extend to only retain the edges with the highest weights in the coarser graph.

i.e. preventing the graph being too deepened. (Too many edges)

3.1.2 Balancing partitions

After the graph partitioning, DistDGL manages two sets of IDs for vertex/edge.

1. Global vertex/edge ID
   to identify vertices/edges among cluster.

1. Local vertex/edge ID
   to efficiently locate vertices/edges in the partition.

3.1.2 Balancing partitions

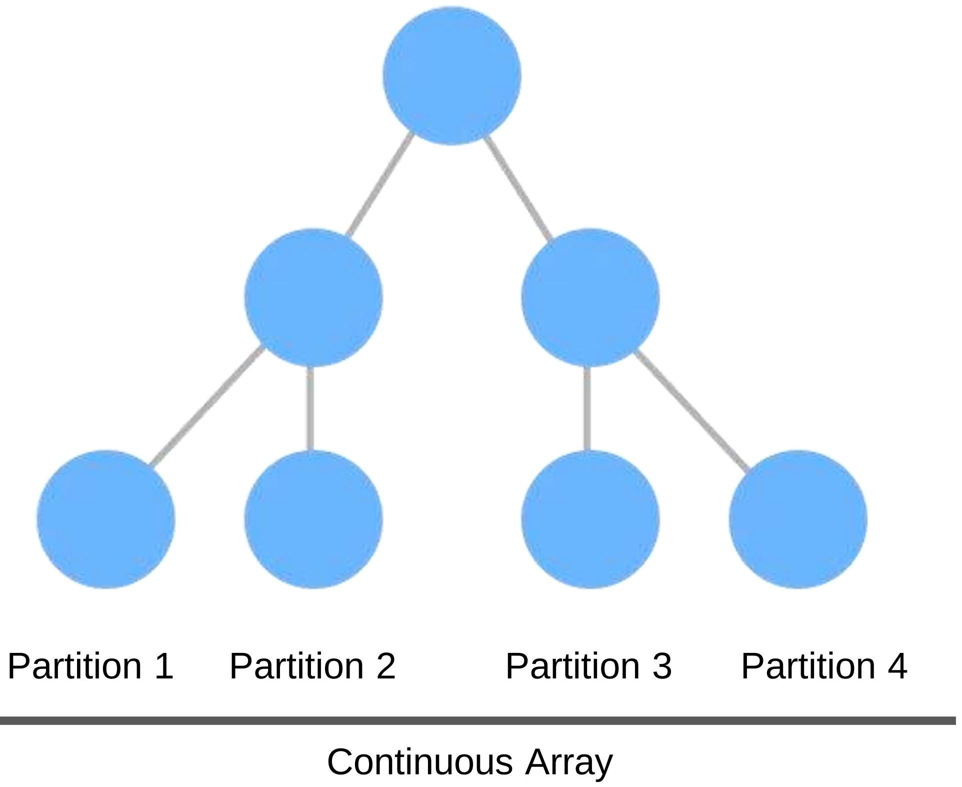
At final step, to save memory for the mapping between global and local IDs, DistDGL relabels vertex and edge IDs

to ensure that all IDs of core vertices/edges in a partition fall into a contiguous ID range.

This relabeling allows:

1. mapping a global ID is binary lookup in a very small array
2. mapping a global ID to a local ID is a simple subtraction operation.

3.1.2 Balancing partitions



1. Leaf nodes have local ID range for each partition.
2. All leaf nodes are on successive array.
3. To obtain global ID from local ID, subtract length of global ID range of front partitions

3.2 Distributed Key-Value Store

Despite the graph partitioning with densely connected vertices,

still need to read data from remote partitions.
e.g. Sparse graphs with vertices in other partitions.

Accessing vertex/edge features usually accounts for the majority of communication in GNN distributed training.

To simplify the data access on other machines,

DistDGL developed a distributed in-memory key-value store (KVStore) dedicated to manage

1. Vertex/Edge features
2. Vertex Embeddings.

3.2 Distributed Key-Value Store

KVStore is specialized for

1. better co-location of node/edge features (in both KVStore and partitions)
2. faster network access
3. efficient updates on sparse embeddings

DistDGL flexibly aligns with graph partitions in each machine
with separated partition policies for each machine.

3.2 Distributed Key-Value Store

Key optimization of KVStore is to use Shared Memory.

Thanks to co-location of data/computation within partitions,
most of data access to KVStore is occurred on local machine.

Thus, instead of IPC,
the KVStore server shares data with the trainer processes via shared memory,

without IPC-related overhead.

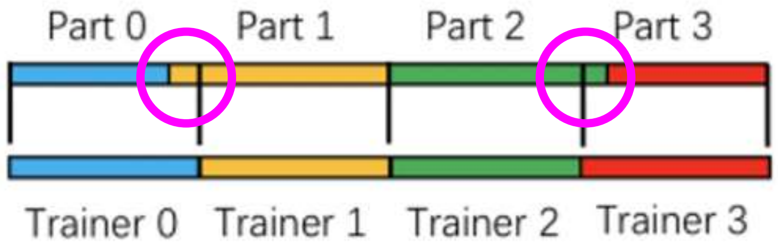
Also, for fast networks (e.g. Infiniband),
DistDGL developed an optimized RPC framework.

3.3 Distributed Sampler

Sampling procedure:

1. The trainer issues sampling requests using the target vertices in the current partition.
2. The requests are dispatched to the machines according to the core vertex assignment (from Graph partitioning)
3. When received, sampler servers call DGL's sampling operators on the local partition.
4. Transmit the result back to the trainer process.
5. The trainer aggregates all the results to generate a mini-batch.

3.4 Mini-batch Trainer



To balance the computation in each trainer,
DistDGL uses a two-level strategy to split the training set evenly

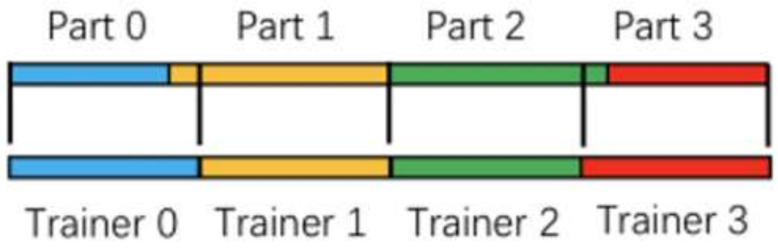
We MUST ensure that each trainer has the same number of training samples. (#)

To do so:

1. Evenly split the training samples based on their IDs
2. Assign the ID range to a machine
   whose graph partition has the largest overlap with the ID range.

This is viable thanks to ID relabeling.

3.4 Mini-batch Trainer



This misalignment is a tradeoff between

1. Load balancing
2. Data locality

In practice, as long as the graph partitioning balances the number of training samples between partitions,
the tradeoff is negligible.

3.4 Mini-batch Trainer

For distributed CPU training,
DistDGL parallelizes the computation with both

1. Multithreading
   for operator computations
   e.g. Matrix multiplication

2. Multiprocessing
   for cluster with NUMA architecture.

NOTE: More trainer processes result in more communication overhead for model parameter updates. (Tradeoff)

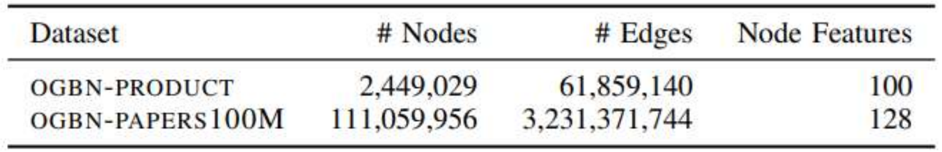
4. Evaluation

Basis for evaluation:

1. Can DistDGL train GNNs on large graphs on cluster in scale?
2. Can DistDGL effectively increase the data locality for GNN training?
3. Can load balancing strategies effectively balance the workloads in cluster?

4.1. Benchmarks / Settings

TABLE I: Dataset statistics from the Open Graph Benchmark [33].



Dataset	# Nodes	# Edges	Node Features
OGBN-PRODUCT	2,449,029	61,859,140	100
OGBN-PAPERS100M	111,059,956	3,231,371,744	128

Benchmarks

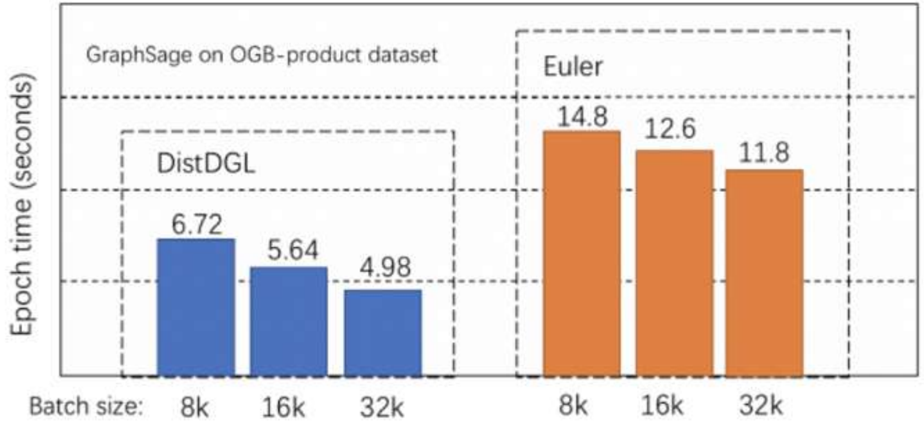
GraphSAGE model
on two Open Graph Benchmark (OGB) datasets.

Settings

A cluster of eight AWS EC2 m5n.24xlarge instances (96 VCPU, 384GB RAM each) connected by a 100Gbps network.

4.2. Comparison to other distributed GNN frameworks

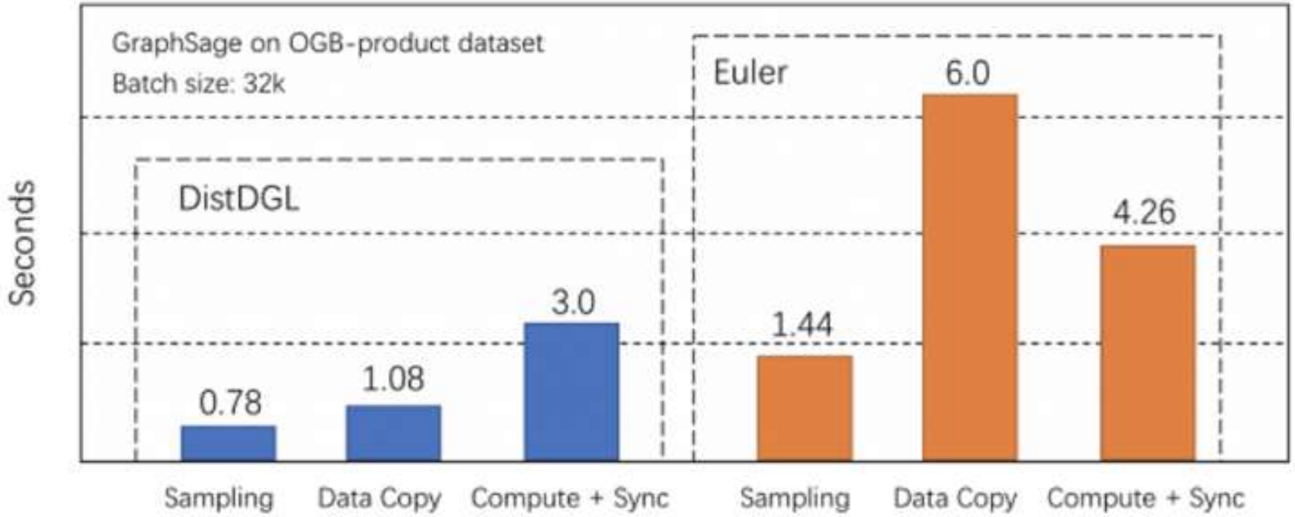
Comparison of Training speed with Euler.



(a) The overall runtime per epoch with different global batch sizes.

DistDGL gets 2.2× speedup over Euler in all different batch sizes

4.2. Comparison to other distributed GNN frameworks



(b) The breakdown of epoch runtime for the batch size of 32K.

4.2.1. Data Copy

The main advantage of DistDGL (5x speed up over Euler)

This is thanks to

1. METIS
   for Graph partitioning with minimal edge cuts.
2. Co-location of partition data within trainers
   for Network communication reduction.

Speed of data copy:
(Nearly) Local memory copy of DistDGL <<< TCP/IP (RPC) of Euler.

4.2.2. Sampling / Implementation

Sampling (2x speed up over Euler)

Thanks to DistDGL samples majority of vertices and edges from the local partition

to generate mini-batches.

Implementation

DistDGL relies on DGL and PyTorch
which are slightly faster than Tensorflow of Euler.

But, Batch computation and Gradient Synchronization are coupled in PyTorch.

4.3. Comparison of Sparse Embedding performance

Sparse embedding performance is evaluated with modified GraphSage model.

Comparison target is PyTroch Sparse Embedding.

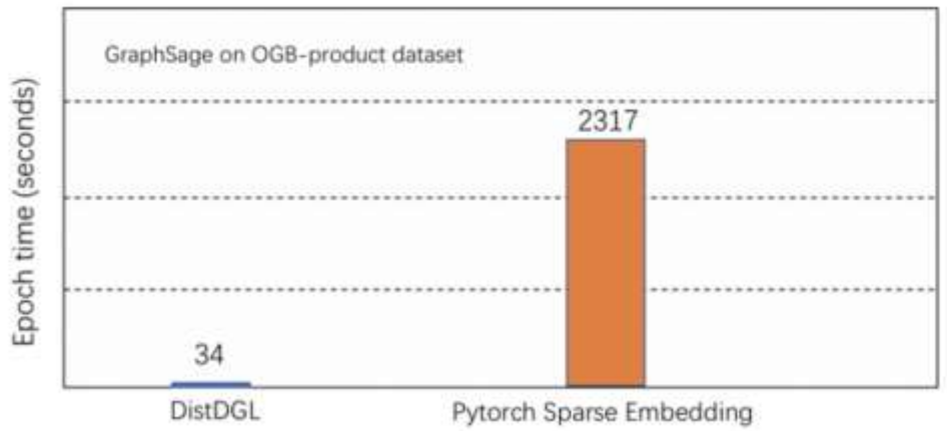


Fig. 7: The GraphSage model with DistDGL's and Pyotch's sparse Embedding on the OGBN-PRODUCT graph.

4.3. Comparison of Sparse Embedding performance

DistDGL gets almost 70x speed up.

This is thanks to efficient update of sparse embedding via KVStore.

In contrast, PyTorch (`DistributedDataParallel` Module) requires the gradient tensor must have the same shape,

deteriorating performance by padding the gradient tensor.

4.4. Scalability

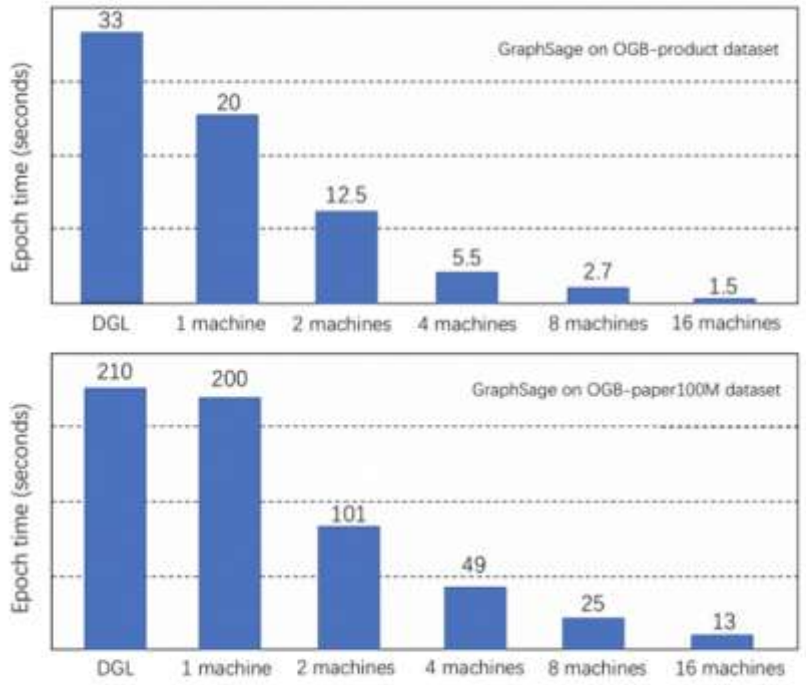


Fig. 8: DistDGL achieves linear speedup w.r.t. the number of machines.

DistDGL achieves a linear speedup as the number of machines increases.

This indicates that DistDGL's optimization

1. well handles network communication.
2. well balances the workloads.

4.5. Ablation study for Graph partitioning

Comparison of DistDGL’s Graph partitioning (multi-constraints METIS) with two alternatives:

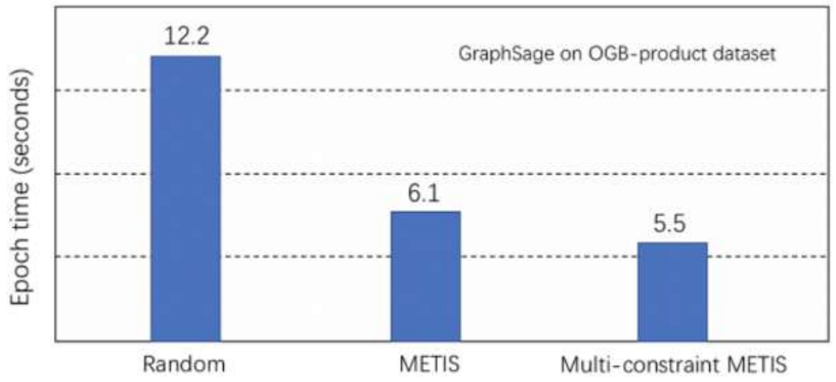
1. Random graph partitioning
2. Default METIS partitioning (without multi-constraints)

4.5. Ablation study for Graph partitioning

In OGB-product dataset,

Default METIS partitioning performs well compared with random partitioning, which suggests that METIS' network communication reduction (due to partitioning) works well.

Multi-constraint METIS gains 4% improvement over Default METIS.

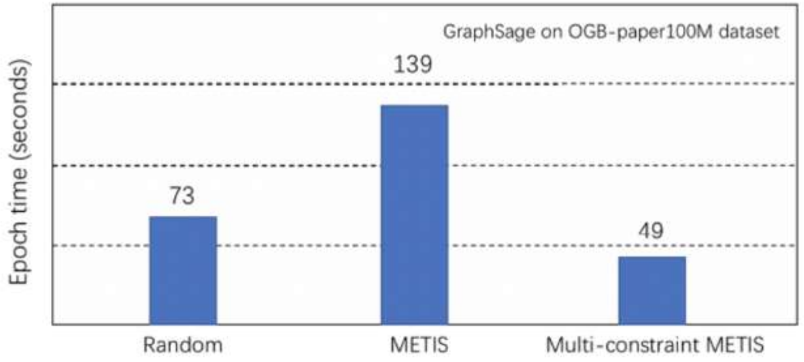


4.5. Ablation study for Graph partitioning

In OGB-paper dataset,
where data is highly imbalanced,

Default METIS performs worse than Random partitioning.

This suggests that Multi-constraint METIS' ability to further balancing partitions (Load Balancing) plays an important role in imbalanced dataset.


GraphSage on OGB-paper100M dataset
Epoch time (seconds)
Random
METIS
Multi-constraint METIS