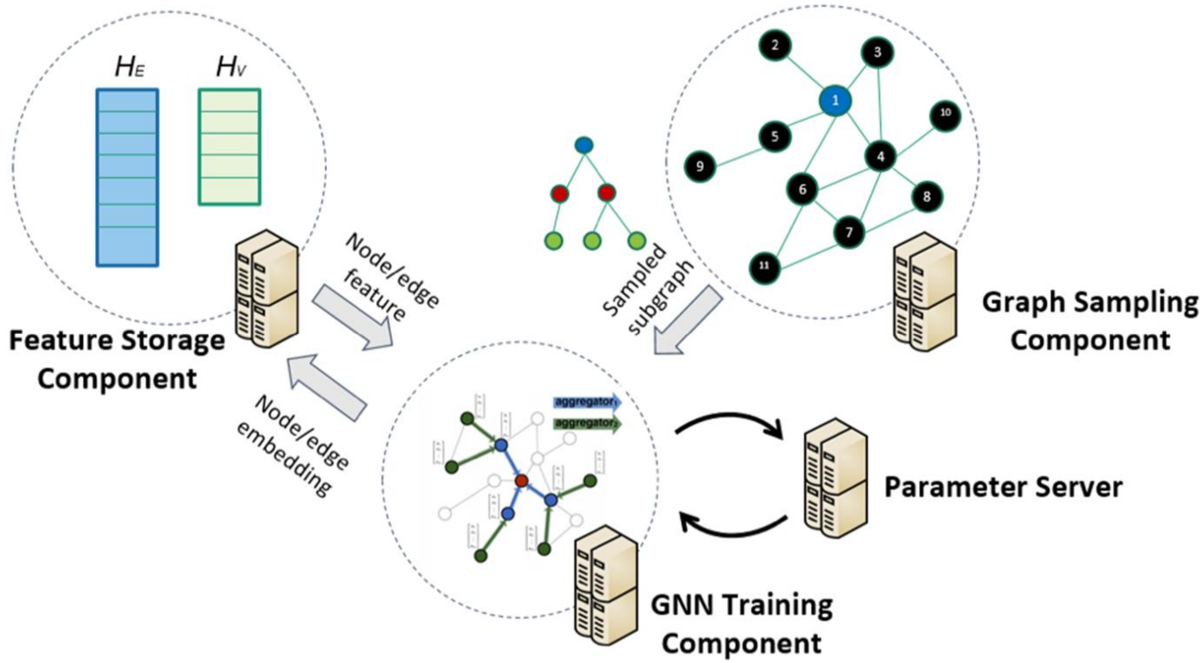
Distributed training in DGL

Da Zheng

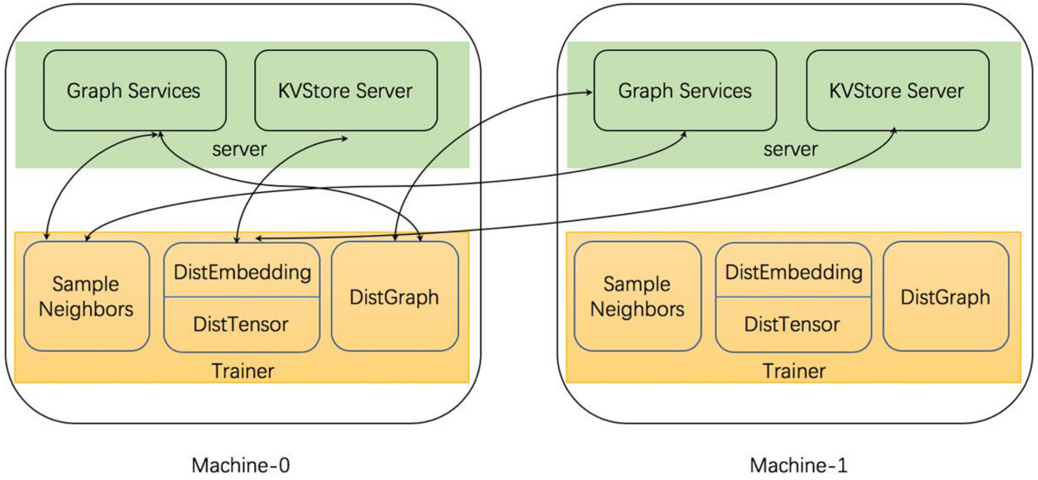
Overview

- Fully distributed
- Locality-aware partition
- Strike for linear speedup
- Minimize code change



Distributed architecture

- All machines run servers and trainers.
- DGL partitions a graph and one machine is responsible for one partition.
- Trainer processes access distributed data via:
    - *DistGraph* for both graph structure and node/edge data.
    - *DistTensor* and *DistEmbedding* accesses distributed tensors.
    - *sample_neighbors* samples subgraphs.



Overview of distributed training code

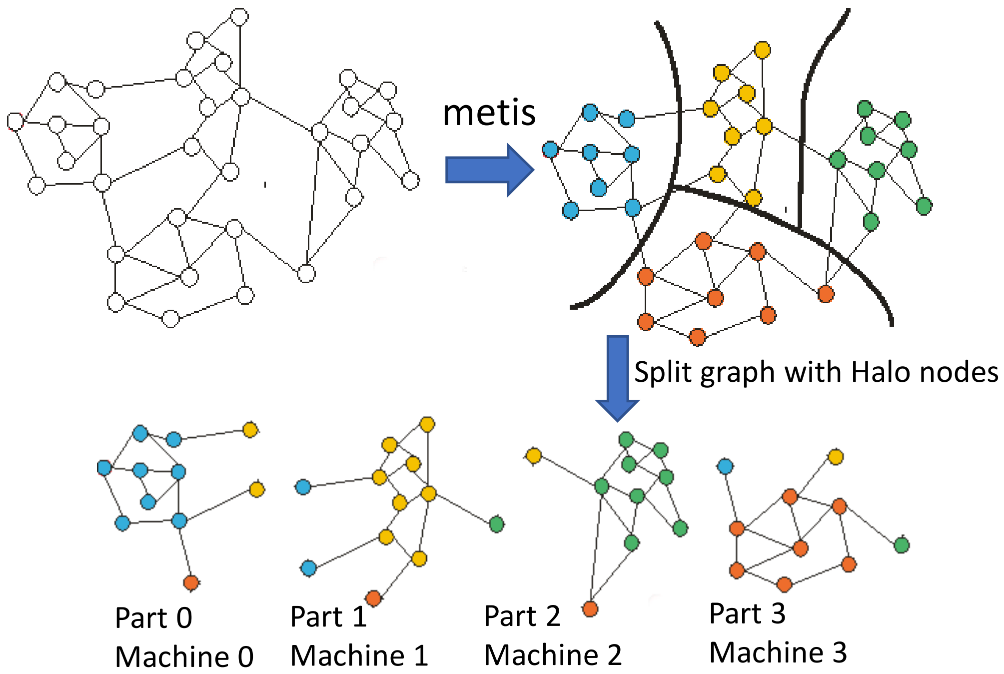
- Little modification is required for distributed training.
  - Call *initialize* at the very beginning.
  - Use *DistGraph* for the distributed graph.
  - Use *node_split* to get a training subset for the trainer process.

```
import dgl
import torch as th

dgl.distributed.initialize(ip_config, num_workers=num_workers)
th.distributed.init_process_group(backend='gloo')
g = dgl.distributed.DistGraph('graph_name')
train_nid = dgl.distributed.node_split(g.ndata['train_mask'], g.get_partition_book())
sampler = dgl.dataloading.MultiLayerNeighborSampler([10, 25])
train_dataloader = dgl.dataloading.NodeDataLoader(g, train_nid,
                                                  sampler, batch_size=1024,
                                                  shuffle=True,drop_last=False)

# Define model and optimizer
model = SAGE(in_feats, num_hidden, n_classes, num_layers, F.relu, dropout)
model = th.nn.parallel.DistributedDataParallel(model)
loss_fcn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=args.lr)

for epoch in range(args.num_epochs):
    for step, blocks in enumerate(dataloader):
        batch_inputs, batch_labels = load_subtensor(g, blocks[0].srcdata[dgl.NID],
                                                    blocks[-1].dstdata[dgl.NID])
        batch_pred = model(blocks, batch_inputs)
        loss = loss_fcn(batch_pred, batch_labels)
        optimizer.zero_grad()
        loss.backward()

        # Aggregate gradients in multiple nodes.
        for param in model.parameters():
            if param.requires_grad and param.grad is not None:
                th.distributed.all_reduce(param.grad.data,
                                          op=th.distributed.ReduceOp.SUM)
                param.grad.data /= dgl.distributed.get_num_clients()
        optimizer.step()
```

Data preprocessing

Graph partitioning

- Two supported partitioning algorithms:
  - Metis
  - Random
- Metis graph partitioning assigns nodes to partitions.
  - Minimize edge cuts
- Split the graph into subgraphs (partitions); one partition per machine.

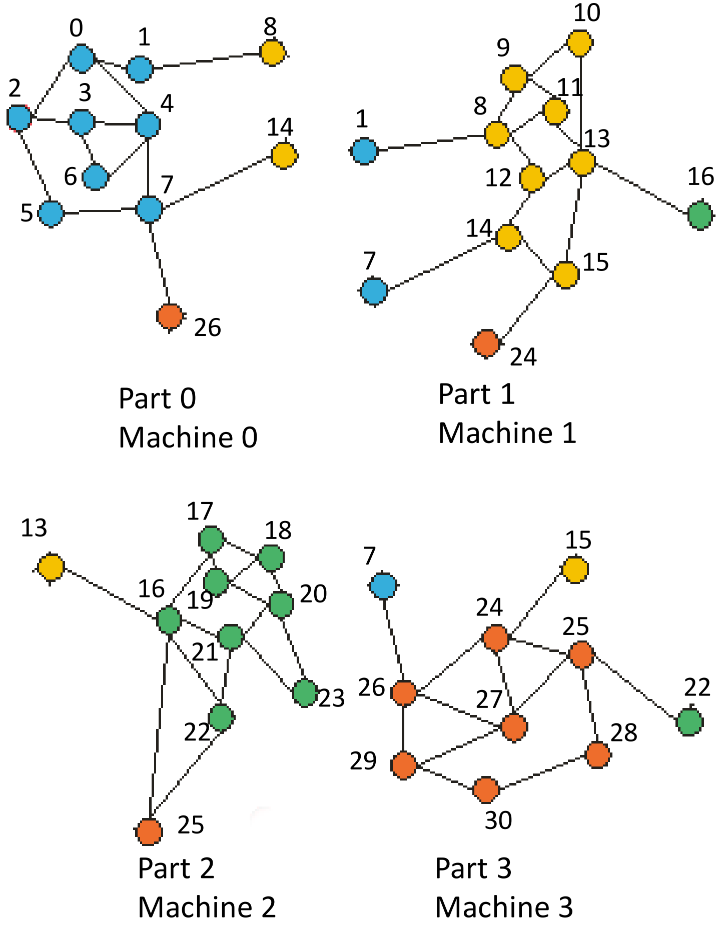


Load balancing with Metis

- Balancing criteria:
  - The number of nodes for each node type
  - In-degrees of nodes for each node type
- Balance the sizes of training, validation and testing
  - Treat nodes in training, validation and testing sets as different node types.

Id relabeling

- Relabel node Ids and edge Ids:
  - All node/edge Ids in a partition fall in a contiguous range.
- Why relabeling?
  - Mapping a node/edge to a partition requires large memory (#nodes/#edges in a graph).
  - Relabeling allows a small array (#partitions) to map node/edge to a partition.
- The distributed code doesn't care about Id relabeling.
  - Node/Edge features are reshuffled accordingly.



Tools for graph partitioning

- DGL provides API *dgl.distributed.partition_graph* for graph partitioning:
  - Load a graph to DGL in a customized way.
  - More controls for load balancing.

Partition results

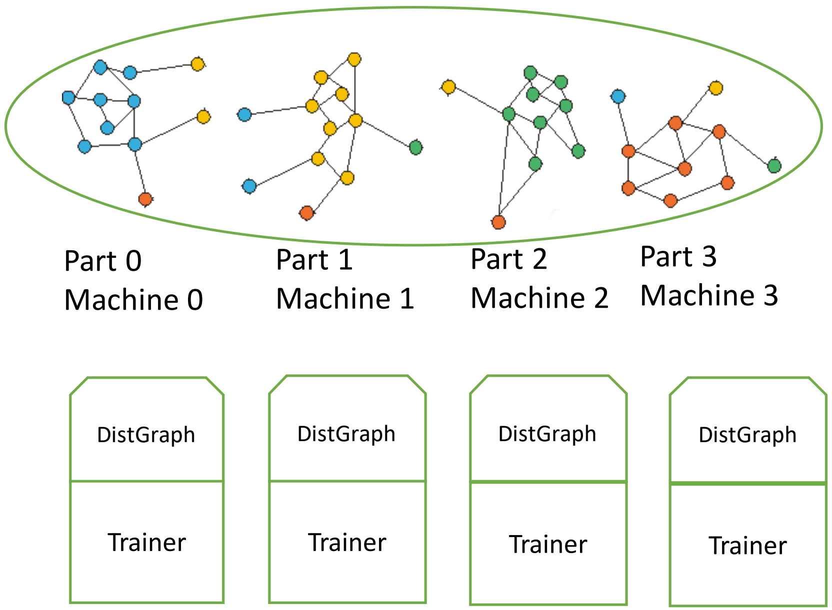
- A JSON file to store the partition configurations.
- Node/edge mapping file (optional)
  - If node/edge Id relabeling is enabled, there is no node/edge mapping files.
- Data for each partition (in the DGL format)
  - Graph structure of a partition.
  - Features of nodes that belong to the partition.
  - Features of edges that belong to the partition.

```
data_root_dir/
  |-- graph_name.json
  |-- node_map.npy
  |-- edge_map.npy
  |-- part0/
      |-- node_feats.dgl
      |-- edge_feats.dgl
      |-- graph.dgl
  |-- part1/
      |-- node_feats.dgl
      |-- edge_feats.dgl
      |-- graph.dgl
....
```

Distributed components for trainer processes

DistGraph

- A Python class in the trainer process for accessing the graph structure and node/edge features in the cluster of machines.
- Execution mode: distributed vs. standalone
  - Standalone: model development and testing.
  - Distributed: run code in a cluster.
  - No code change when switching between standalone and distributed.



DistGraph creation

- Distributed mode:
  - Graph data is loaded by DGL servers.
  - DistGraph connects to the DGL servers.

```
>>> import dgl
>>> g = dgl.distributed.DistGraph('graph_name')
```

- Standalone mode:
  - data is loaded by DistGraph.

```
>>> import dgl
>>> g = dgl.distributed.DistGraph('graph_name', part_config='data/graph_name.json')
```

Data access in DistGraph

- Access graph structure:
    - The access to the graph structure is very limited.

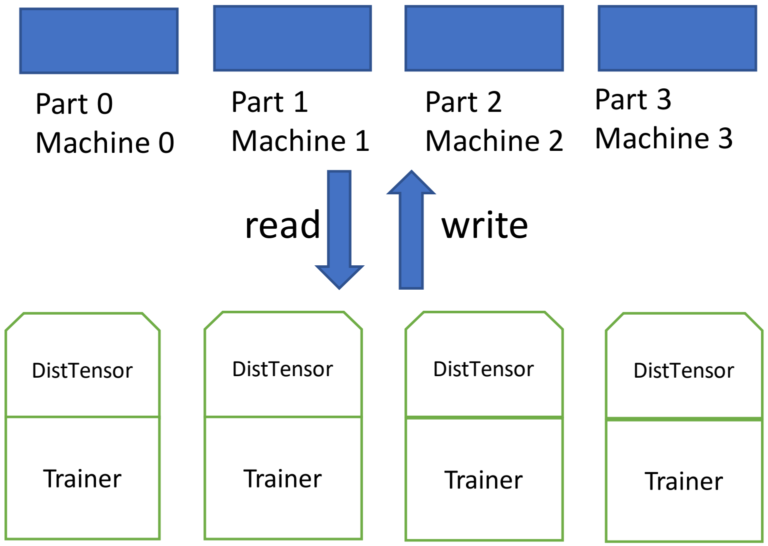
```
>>> print(g.number_of_nodes())
```

- Access node/edge data:
    - ndata
    - edata

```
>>> g.ndata['train_mask']
<dgl.distributed.dist_graph.DistTensor at 0x7fec820937b8>
>>> g.ndata['train_mask'][0]
tensor([1], dtype=torch.uint8)
```

Distributed tensor

- dgl.distributed.DistTensor accesses tensors sharded in a cluster of machines.
- Data is accessed with global row Ids.



Operations on DistTensor

- Create DistTensor
  - Allocate memory if the tensor identified by the tensor name doesn’t exist in the cluster.
  - Reuse the tensor if the tensor identified by the tensor name exists.
  - This is a synchronized operation.

```
>>> tensor = dgl.distributed.DistTensor((g.number_of_nodes(), 10), th.float32, ‘test’)
```

- Read DistTensor: copy specified rows to the local process.

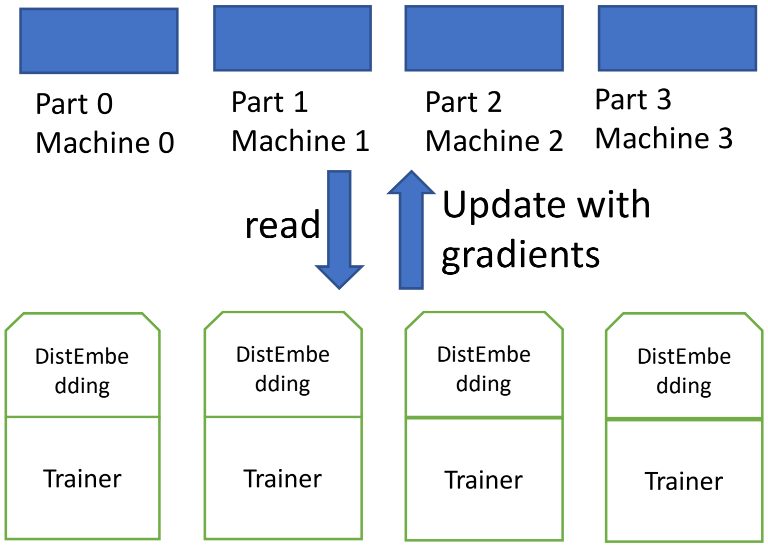
```
>>> data = tensor[0:100]
```

- Write DistTensor: write data to the specified rows.

```
>>> tensor[0:100] = data
```

Distributed Embedding

- Some models (e.g., DeepWalk) require learnable embeddings.
- dgl.distributed.DistEmbedding accesses embeddings sharded in a cluster of machines.
- Data is accessed with global row Ids.
- Update embeddings with DGL's sparse optimizer.



Operations on Distributed Embeddings

- Create DistEmbedding
  - Allocate memory if the embeddings identified by the name doesn't exist in the cluster.
  - Reuse the embeddings if the embeddings identified by the tensor name exists.
  - This is a synchronized operation.

```
>>> emb = dgl.distributed.DistEmbedding(g.number_of_nodes(), 10, 'test')
```

- Read DistEmbedding: copy specified rows to the local process.

```
>>> data = emb(np.arrange(100))
```

Operations on Distributed Embeddings

- Update embeddings with DGL's sparse optimizer.
  - Currently, DGL provides dgl.distributed.SparseAdagrad.

```
>>> optimizer = dgl.distributed.SparseAdagrad([emb], lr=0.001)
>>> feats = emb([0,1,2,3])
>>> loss = th.sum(feats + 1)
>>> loss.backward()
>>> optimizer.step()
```

  - When *step* is invoked, the embeddings of row 0, 1, 2, 3 are updated by *SparseAdagrad*.

Graph sampling

- DGL provides distributed sampling
  - Sample seed nodes/edges
  - Sample neighbors: dgl.distributed.sample_neighbors
- The high-level sampling APIs (e.g., NodeDataLoader) work for both DGLGraph and DistGraph.

```
>>> sampler = dgl.dataloading.MultiLayerNeighborSampler([10, 25])
>>> train_dataloader = dgl.dataloading.NodeDataLoader(
        g, train_nid, sampler,
        batch_size=1024,
        shuffle=True,
        drop_last=False
)
```

Split training, validation and test set

- Store the training/validation/test set in global mask tensors in preprocessing.

```
>>> g.ndata['train_mask']
```

- Get nodes/edges from the training set in a trainer process.

```
>>> train_nid = dgl.distributed.node_split(g.ndata['train_mask'])
```

Invoke distributed training

Copy partitions to the cluster

- DGL provides a data copy script to copy partitions
  - It requires ssh passwordless access between machines.
  - Copy a partition to the right machine.
  - Update the partition configuration file automatically.

```
python3 copy_files.py --ip_config ip_config.txt \
--workspace ~/graphsage/ \
--rel_data_path 4part_data --part_config 4part_data/ogb-product.json
```

Launch distributed training

- DGL provides a launch script to run distributed training and inference in a cluster of machines.
  - It requires ssh passwordless access between machines.

```
python3 ~/dgl/tools/launch.py \
--workspace ~/graphsage/ \
--num_clients 2 \
--part_config data/ogb-product.json \
--ip_config ip_config.txt \
"python3 train_dist.py --graph-name ogb-product --ip_config ip_config.txt --num-epochs 30
--batch-size 1000 --lr 0.1"
```

- The script launches the servers and trainers on the cluster of machines.

Next

- Jupyter Notebook to demonstrate graph partitioning.
- Jupyter Notebook to demonstrate the basic operations on distributed components (DistGraph, DistTensor, etc).
- Jupyter Notebook to demonstrate distributed GraphSage for node classification.
- Jupyter Notebook to demonstrate distributed GraphSage with embeddings for node classification.
- Convert the Notebooks into training scripts and run distributed training on a cluster.