7.4 Advanced Graph Partitioning

The chapter covers some of the advanced topics for graph partitioning.

METIS partition algorithm

METIS is a state-of-the-art graph partitioning algorithm that can generate partitions with minimal number of cross-partition edges, making it suitable for distributed message passing where the amount of network communication is proportional to the number of cross-partition edges. DGL has integrated METIS as the default partitioning algorithm in its `dgl.distributed.partition_graph()` API.

Output format

Regardless of the partitioning algorithm in use, the partitioned results are stored in data files organized as follows:

```
data_root_dir/
  |-- graph_name.json       # partition configuration file in JSON
  |-- part0/                # data for partition 0
  |  |-- node_feats.dgl     # node features stored in binary format
  |  |-- edge_feats.dgl     # edge features stored in binary format
  |  |-- graph.dgl          # graph structure of this partition stored in binary format
  |
  |-- part1/                # data for partition 1
  |  |-- node_feats.dgl
  |  |-- edge_feats.dgl
  |  |-- graph.dgl
  |
  |-- ...                   # data for other partitions
```

When distributed to a cluster, the metadata JSON should be copied to all the machines while the `partX` folders should be dispatched accordingly.

DGL provides a `dgl.distributed.load_partition()` function to load one partition for inspection.

```
>>> import dgl
>>> # load partition 0
>>> part_data = dgl.distributed.load_partition('data_root_dir/graph_name.json', 0)
>>> g, nfeat, efeat, partition_book, graph_name, ntypes, etypes = part_data  # unpack
>>> print(g)
Graph(num_nodes=966043, num_edges=34270118,
      ndata_schemes={'orig_id': Scheme(shape=(), dtype=torch.int64),
                     'part_id': Scheme(shape=(), dtype=torch.int64),
                     '_ID': Scheme(shape=(), dtype=torch.int64),
                     'inner_node': Scheme(shape=(), dtype=torch.int32)}
      edata_schemes={'_ID': Scheme(shape=(), dtype=torch.int64),
                     'inner_edge': Scheme(shape=(), dtype=torch.int8),
                     'orig_id': Scheme(shape=(), dtype=torch.int64)})
```

As mentioned in the `ID mapping`_ section, each partition carries auxiliary information saved as ndata or edata such as original node/edge IDs, partition IDs, etc. Each partition not only saves nodes/edges it owns, but also includes node/edges that are adjacent to the partition (called **HALO** nodes/edges). The `inner_node` and `inner_edge` indicate whether a node/edge truely belongs to the partition (value is `True`) or is a HALO node/edge (value is `False`).

The `load_partition()` function loads all data at once. Users can load features or the partition book using the `dgl.distributed.load_partition_feats()` and `dgl.distributed.load_partition_book()` APIs respectively.

Parallel METIS partitioning

For massive graphs where parallel preprocessing is desired, DGL supports ParMETIS as one of the choices of partitioning algorithms.

Note

Because ParMETIS does not support heterogeneous graph, users need to conduct ID conversion before and after running ParMETIS. Check out chapter 7.5 Heterogeneous Graph Under The Hood for explanation.

Note

Please make sure that the input graph to ParMETIS does not have duplicate edges (or parallel edges) and self-loop edges.

ParMETIS Installation

ParMETIS requires METIS and GKLib. Please follow the instructions here to compile and install GKLib. For compiling and install METIS, please follow the instructions below to clone METIS with GIT and compile it with int64 support.

```
git clone https://github.com/KarypisLab/METIS.git
make config shared=1 cc=gcc prefix=~/local i64=1
make install
```

For now, we need to compile and install ParMETIS manually. We clone the DGL branch of ParMETIS as follows:

```
git clone --branch dgl https://github.com/KarypisLab/ParMETIS.git
```

Then compile and install ParMETIS.

```
make config cc=mpicc prefix=~/local
make install
```

Before running ParMETIS, we need to set two environment variables: `PATH` and `LD_LIBRARY_PATH`.

```
export PATH=$PATH:$HOME/local/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/local/lib/
```

Input format

Note

As a prerequisite, read chapter guide-distributed-hetero to understand how DGL organize heterogeneous graph for distributed training.

The input graph for ParMETIS is stored in three files with the following names: `xxx_nodes.txt`, `xxx_edges.txt` and `xxx_stats.txt`, where `xxx` is a graph name.

Each row in `xxx_nodes.txt` stores the information of a node. Row ID is also the *homogeneous* ID of a node, e.g., row 0 is for node 0; row 1 is for node 1, etc. Each row has the following format:

```
<node_type_id> <node_weight_list> <type_wise_node_id>
```

All fields are separated by whitespace:

- `<node_type_id>` is an integer starting from 0. Each node type is mapped to an integer. For a homogeneous graph, its value is always 0.
- `<node_weight_list>` are integers (separated by whitespace) that indicate the node weights used by ParMETIS to balance graph partitions. For homogeneous graphs, the list has only one integer while for heterogeneous graphs with T node types, the list should has T integers. If the node belongs to node type t, then all the integers except the t^{th} one are zero; the t^{th} integer is the weight of that node. ParMETIS will try to balance the total node weight of each partition. For heterogeneous graph, it will try to distribute nodes of the same type to all partitions. The recommended node weights are 1 for balancing the number of nodes in each partition or node degrees for balancing the number of edges in each partition.
- `<type_wise_node_id>` is an integer representing the node ID in its own type.

Below shows an example of a node file for a heterogeneous graph with two node types. Node type 0 has three nodes; node type 1 has four nodes. It uses two node weights to ensure that ParMETIS will generate partitions with roughly the same number of nodes for type 0 and the same number of nodes for type 1.

```
0 1 0 0
0 1 0 1
0 1 0 2
1 0 1 0
1 0 1 1
1 0 1 2
1 0 1 3
```

Similarly, each row in `xxx_edges.txt` stores the information of an edge. Row ID is also the *homogeneous* ID of an edge, e.g., row 0 is for edge 0; row 1 is for edge 1, etc. Each row has the following format:

```
<src_node_id> <dst_node_id> <type_wise_edge_id> <edge_type_id>
```

All fields are separated by whitespace:

- `<src_node_id>` is the *homogeneous* ID of the source node.
- `<dst_node_id>` is the *homogeneous* ID of the destination node.
- `<type_wise_edge_id>` is the edge ID for the edge type.
- `<edge_type_id>` is an integer starting from 0. Each edge type is mapped to an integer. For a homogeneous graph, its value is always 0.

`xxx_stats.txt` stores some basic statistics of the graph. It has only one line with three fields separated by whitespace:

```
<num_nodes> <num_edges> <total_node_weights>
```

- `num_nodes` stores the total number of nodes regardless of node types.
- `num_edges` stores the total number of edges regardless of edge types.
- `total_node_weights` stores the number of node weights in the node file.

Run ParMETIS and output format

ParMETIS contains a command called `pm_dglpart`, which loads the graph stored in the three files from the machine where `pm_dglpart` is invoked, distributes data to all machines in the cluster and invokes ParMETIS to partition the graph. When it completes, it generates three files for each partition: `p<part_id>-xxx_nodes.txt`, `p<part_id>-xxx_edges.txt`, `p<part_id>-xxx_stats.txt`.

> **Note**
>
> ParMETIS reassigns IDs to nodes during the partitioning. After ID reassignment, the nodes in a partition are assigned with contiguous IDs; furthermore, the nodes of the same type are assigned with contiguous IDs.

`p<part_id>-xxx_nodes.txt` stores the node data of the partition. Each row represents a node with the following fields:

```
<node_id> <node_type_id> <node_weight_list> <type_wise_node_id>
```

- `<node_id>` is the *homogeneous* node ID after ID reassignment.
- `<node_type_id>` is the node type ID.
- `<node_weight_list>` is the node weight used by ParMETIS (copied from the input file).
- `<type_wise_node_id>` is an integer representing the node ID in its own type.

`p<part_id>-xxx_edges.txt` stores the edge data of the partition. Each row represents an edge with the following fields:

```
<src_id> <dst_id> <orig_src_id> <orig_dst_id> <type_wise_edge_id> <edge_type_id>
```

- `<src_id>` is the *homogeneous* ID of the source node after ID reassignment.
- `<dst_id>` is the *homogeneous* ID of the destination node after ID reassignment.
- `<orig_src_id>` is the *homogeneous* ID of the source node in the input graph.
- `<orig_dst_id>` is the *homogeneous* ID of the destination node in the input graph.
- `<type_wise_edge_id>` is the edge ID in its own type.

- `<edge_type_id>` is the edge type ID.

When invoking `pm_dglpart`, the three input files: `xxx_nodes.txt`, `xxx_edges.txt`, `xxx_stats.txt` should be located in the directory where `pm_dglpart` runs. The following command run four ParMETIS processes to partition the graph named `xxx` into eight partitions (each process handles two partitions).

```
mpirun -np 4 pm_dglpart xxx 2
```

The output files from ParMETIS then need to be converted to the partition assignment format to in order to run subsequent preprocessing steps.