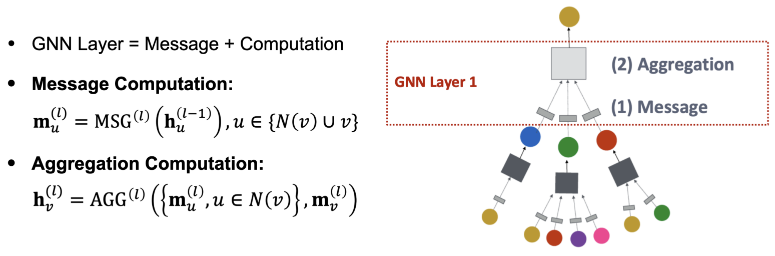
ID2203 Project Course - Distributed Graph Neural Networks Training

Sonia Horchidan, sfhor@kth.se

1 Project Description

Graph Neural Networks (GNNs) are a powerful Deep Learning paradigm designed for data with inherent graph structures. They leverage message-passing mechanisms to aggregate information (i.e., features) from neighboring nodes, enabling tasks like node classification, graph property prediction, and link prediction. Their flexibility has led to diverse applications across various domains, from social network analysis to protein modeling.
Training GNNs is, however, a complex process. In simple terms, the novelty of training a GNN consists of its message-passing framework, which follows, at a high level, the next steps:

1. **Message Creation** - For each node, its features are combined with the features of its neighboring nodes (via the edges) through a "message function" (denoted MSG in figure 1). This function typically involves a neural network layer that transforms the combined information. The aggregation is performed at the node's k-hop neighborhood level.
2. **Message Aggregation** - The messages received by a node from a sample of its neighbors are aggregated using an "aggregation function" (denoted AGG in figure 1). This function combines the information from multiple messages into a single representation for the receiving node. Common aggregation functions include summation, averaging, or more complex neural network layers that allow for selective aggregation based on importance.
3. **Node Update** - The aggregated message is combined with the original features of the receiving node through an "update function." This function typically involves another neural network layer that transforms the combined information into an updated representation for the node. The update function incorporates the information from neighbors, allowing the node's representation to evolve based on its context in the graph.
4. **Iteration** - These steps (message creation, aggregation, update) are repeated for multiple iterations, allowing information to propagate further through the graph



GNN Layer = Message + Computation
Message Computation:
$\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right), u \in \{N(v) \cup v\}$
Aggregation Computation:
$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}, \mathbf{m}_v^{(l)}\right)$
GNN Layer 1
(2) Aggregation
(1) Message

Figure 1: Message passing and aggregation in standard GNNs[1].

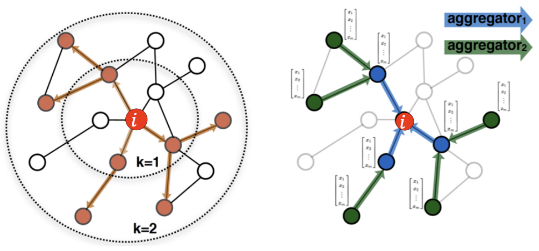


Figure 2: k-hop neighbourhood aggregation[1].

and incorporate information from more distant neighbors, as it can be noted in figure 2. The hop number (k) determines the maximum depth you want to explore the graph from the target node. Parameters ($\delta_0, ..., \delta_{k-1}$) can be set to specify the sampling strategy used at each hop of the aggregation. The node will aggregate δ_0 messages from neighbors at hop 1, δ_1 messages from neighbors at hop 2, and so on.

The state-of-the-art in distributed GNN training is to instrument the epochs using a Bulk Synchronous Parallel (BSP) type model, where a barrier synchronization separates the steps. These steps are also known as supersteps. Fully asynchronous training methods also exist, but the model might struggle to converge due to slow workers and outdated information.
In this project, we will take inspiration from the GNN's training process but focus solely on the message-passing communication steps presented above. The goal is to implement and test a simple, partitioned, distributed, in-memory graph storage that supports k-hop neighborhood queries based on message-passing. We assume that each node stores one integer value as its feature for simplicity. Assume the MSG function is the identity function, while the aggregation function AGG is summation.
You have significant freedom in your choice of implementation, programming language, and libraries. However, we strongly encourage you to use C++.

[1] The figures were taken from Stanford's Winter 2021 course CS224W: Machine Learning with Graphs.

1.1 Assumptions

1. The workers cannot fail. Similarly, no new workers can join the network. You can start four workers and assume they will be up and running throughout the whole execution of the program.
2. The graph is static and needs to be loaded at startup (bootstrapping). You can assume each worker has a graph partition ready to be read from the disk. The worker should load all the contents of this file. That is, the worker with ID 1 should read partition1.csv.
3. We use a hash partitioning scheme. After loading the graph at bootstrap time, you can assume that the graph node with ID n_{id} will be stored by worker $n_{id}\%4$ (given that 4 workers are up and running).
4. The graph uses a vertex-partitioning scheme. The nodes are assigned to partitions, and, as a result, edges might end up connecting vertices assigned to different partitions.
5. No GPUs will be required for this task.

2 Requirements

2.1 Bootstrapping and network communication (5p)

For this task, you have to design the basic infrastructural layers for the graph store. The partitions need to be distributed over the available workers. You need to be able to set up the system and support $GET(n_{id})$ operations, which should return the feature of the node with id n_{id}. Each worker should run a separate process. You are free to use any networking protocol you are familiar with. For the report, describe and motivate all your decisions and tests.

2.2 Neighbourhood queries (5p)

After you have gotten the basic infrastructure working, you should implement k-hop neighborhood operations with the following interface:

$$KHOP\_NEIGHBORHOOD(n_{id}, k, (\delta_0, ..., \delta_{k-1})))$$

For example, assume a graph of 4 nodes: N_1, N_2, N_3, N_4, with $12, 4, 22, 9$ as corresponding features. Assume N_1 and N_3 are direct neighbors of N_2. In this case:

$$KHOP\_NEIGHBOURHOOD(N_2, 1, 2) \text{ should return } (12 + 22) + 4 = 28$$
$$KHOP\_NEIGHBOURHOOD(N_2, 1, 1) \text{ can either return } 12 + 4 = 16 \text{ or } 22 + 4 = 26$$

Keep in mind that the neighborhood might be partitioned across multiple workers. As before, be sure to write test scenarios for the simulator.

2.3 Message-passing-based neighborhood aggregation (5p)

We are now ready to simulate the message-passing-based neighborhood aggregation of GNN training. Modify your code to support a $TRAIN(epochs)$ operation. When initiated, iterates over the nodes and runs $KHOP\_NEIGHBORHOOD$ operations to compute their new weights. The nodes should store a history of aggregation obtained at each iteration. Keep in mind that the aggregation at epoch N needs to use the weights computed at epoch $N - 1$. The $TRAIN$ operation should return the sum of the final weights of all the graph nodes. You might consider some coordination here. How do we decide when one epoch is completed? Alternatively, a fully asynchronous $TRAIN$ operation is also allowed if properly described in the report.

2.4 Marker-based asynchronous "training" (15p)

We can now simulate an asynchronous training process via iterative, asynchronous k-hop neighborhood aggregations. At each epoch, our goal is to update the weights of all the nodes in the graph by aggregating their k-hop neighborhood, given the weights of the node in the neighborhood at the previous epoch. You are allowed to use a fixed value of k here. We will explore an idea inspired by the epoch-based snapshotting algorithm: using markers to enforce the causality between aggregations across supersteps. Each worker needs to start with a random node and insert the marker. The marker has to be disseminated across graph nodes (and across workers when necessary) when the computation on a node is completed. For instance, if one node is done with the computation at iteration 1, it can signal its neighbors that it is ready to go to the next iteration. You should think about the relation between the value of k and the marker dissemination process.

2.5 Benchmarks and performance evaluation (10p)

Lastly, your task is to evaluate the performance and scalability of your implemented graph store. Choose relevant benchmarks that assess key aspects like read throughput, latency for GET operations, neighborhood query execution time, and training convergence speed. Describe your testing methodology, including hardware setup, workload selection, and metrics measured. Analyze the results, identifying performance bottlenecks and potential optimizations. Remember to compare your findings to existing solutions or baselines if applicable, and highlight the trade-offs involved in different design choices.

3 Useful Resources

[1] Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." Advances in neural information processing systems 30 (2017).
[2] Valiant, Leslie G. "A bridging model for parallel computation." Communications of the ACM 33.8 (1990): 103-111.

[3] Chandy, K. Mani, and Leslie Lamport. "Distributed snapshots: Determining global states of distributed systems." ACM Transactions on Computer Systems (TOCS) 3.1 (1985): 63-75.