ID2203 Project Course - Distributed Graph Neural Networks Training

Long Ma, Yining Hou, and Hong Jiang

March 22, 2024

1 Introduction

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 neighbouring nodes, enabling tasks like node classification, graph property prediction, and link prediction.

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 neighbourhood queries based on message-passing. We assume that each vertex stores one integer value as its feature for simplicity. Assume the MSG function is the identity function, while the aggregation function AGG is the summation.

Relevant GitHub links:
`https://gits-15.sys.kth.se/yiningho/id2203-vt24-course-project-graphs`.

2 Design

Assumptions

1. We start four workers and assume they will be up and running throughout the whole execution of the program. The workers cannot fail. Similarly, no new workers can join the network.
2. The graph is static, directed and loaded at startup, each worker has a graph partition ready to be read from the disk, e.g. the worker with ID 1 should read partition_1.txt.
3. We use a hash partitioning scheme. After loading the graph at bootstrap time, the graph vertex with ID nid will be stored by worker `nid%4` (given that 4 workers are up and running).

4. The graph uses a vertex-partitioning scheme. The vertexes are assigned to partitions, and, as a result, edges might end up connecting vertices assigned to different partitions.

5. We found through testing that the graph we need to process is acyclic, so we temporarily assume that the graph we need to process is a directed acyclic graph.

Architectural

We choose the client-server architectural pattern here where one client communicates with 4 worker servers.

- Roles
  - Server: The central component that provides services to clients. It loads the static graph features and hashed partitioning scheme by its ID, and it listens for incoming connections, processes requests from clients and other servers and sends responses back.
  - Client: An entity that requests services from the server. The Client sends requests and receives responses.

- Communication Protocol
  Several communication protocols could be used when exchanging messages among servers and clients, while we chose Socket via TCP and RPC in our implementation.

Communication Protocols

As we mentioned above, TCP Socket and RPC are the protocols we are more familiar with:

- Socket
  A TCP socket is an endpoint for communication between two machines over a TCP/IP network. TCP (Transmission Control Protocol) is one of the core protocols of the Internet Protocol Suite, providing reliable, connection-oriented communication between devices. TCP sockets enable processes running on different devices to establish connections, exchange data, and communicate with each other over a network, and can ensure that the order of sending and receiving of all data in the communication is FIFO.

- RPC
  Remote Procedure Call (RPC) is used to enable a program to execute a procedure (or subroutine) in another address space (commonly on another computer on a shared network).

  In Python, we chose `ThreadedXMLRPCServer`, which can handle requests in parallel, thus improving the server's scalability and responsiveness.

The `ThreadedXMLRPCServer` is created by combining two classes:

- ThreadingMixIn: A mixin class from the socket server module that provides threading capabilities to a server. It makes the server multi-threaded by handling each request in a separate thread. This allows the server to manage multiple requests at the same time without blocking.
- SimpleXMLRPCServer: A class that implements a basic XML-RPC server. XML-RPC is a Remote Procedure Call method that uses XML to encode its calls and HTTP as a transport mechanism.

Partition to Grpah

To more conveniently get all the neighbours of any vertex from the file describing the edges, we use the `NetworkX` library.

- `NetworkX` is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. We use this library to convert the edge files of these directed graphs into graphs and use the library's built-in methods to quickly get all the neighbours of the vertex.

For the marker-based asynchronous training, due to hardware limitations, we simplified the graph and maintained the same properties as the original graph. We wrote a `testgrpah.py` script to verify these properties.

Neighbourhood Aggregation

- Message-Passing-based training: In synchronous training, we wait for each epoch to end and start a new epoch. Therefore, all vertexes are in the same epoch and we can safely retrieve its feature when doing k-hop. In asynchronous training, a different vertex can be in different epochs. We use a shared `epoch_dict` to collect each vertex's epoch and store the history of each vertex's feature. We should only do k-hop when the vertex's neighbours are in the same epoch as itself.
- Marker-based asynchronous training: We combined the Chandy-Lamport algorithm with the Epoch Snapshotting algorithm. Every vertex is independent and only retrieves information from its neighbours. We use a marker to isolate different epochs and trace the snapshot, which is the resulting feature of each vertex in the whole graph.

Termination property

To prove that in a directed acyclic graph (DAG), traversing backwards from all vertexes with out-degree 0 can reach all vertexes in the graph, we can use mathematical induction. We can discuss two scenarios:

- Base case: When there is only one vertex with out-degree 0 in the graph, it is obvious that traversing backwards from this vertex can reach all vertexes in the graph.
- Induction hypothesis: Assume for any directed acyclic graph G, traversing backwards from all vertexes with out-degree 0 can reach all vertexes in the graph.

Now consider a directed acyclic graph G', which is formed by adding a new vertex connected to some vertex in G. Since G is a directed acyclic graph, the vertexes with out-degree 0 in G still exist and are also vertexes with out-degree 0 in G'. According to the induction hypothesis, traversing backwards from all vertexes with out-degree 0 can reach all vertexes in G. Since the newly added vertex is connected to a vertex in the graph, traversing backwards from all vertexes with out-degree 0 will also reach the newly added vertex in G'. Therefore, all vertexes in G' can be reached.
According to mathematical induction, we conclude that in a directed acyclic graph, traversing backwards from all vertexes with out-degree 0 can reach all vertexes in the graph. So, we satisfied Termination properties: if the protocol is initiated by a set of processes that can reach all tasks. So that, eventually every process records its state.

Validity property

Chandy Lamport and Epoch sharpshooting: Since both Chandy Lamport and epoch snapshotting need to ensure that the transmission of information between vertexes is FIFO to achieve validity properties, we need some queues and inboxes to ensure this.
Our TCP Socket can guarantee that each message is FIFO inside the message, but it does not guarantee the order between messages. Therefore, when the vertex socket receives any message, the main thread will put the messages in the message queue by the order received. This ensures that when there are multiple senders, the messages received by the vertex are single-source FIFO.
To ensure that the single source FIFO reads messages sequentially and maximizes efficiency, we create an inbox for each message incoming edge (out-edge of the graph structure) and put all inboxes in a dictionary as a class internal variable of a vertex. We will start a thread when initializing the vertex. This thread can take the messages out of the message queue one by one and put them into the inbox of each edge in the order they are received. We did not put this logic in the main thread because the main thread needs to continue to receive messages.
We also established a thread for each message incoming edge (out-edge of the graph structure) for FIFO processing of all messages in a specific vertex inbox. Using all the structures and logic above, we can guarantee FIFO ordering of all messages and markers transmitted between vertex and vertex. Then using an induction proof method similar to the proof Termination property, we can

prove that the Validity property is always true for all markers in the directed acyclic graph.

3 Implementation

Features

Part 1: Bootstrapping and Network Communication

For this part, we design a basic infrastructure to distribute graph partitions over workers, allowing each to support GET operations for vertex features.
Key Steps:

- Graph Partition Loading: Each worker process will load a graph partition from a TXT file named $partition\langle ID\rangle$.txt, where ID is the worker's ID.
- GET Operation: Implement a GET operation that allows querying a vertex's feature by its ID. This requires determining the responsible worker based on the vertex ID and sending a request to that worker if the vertex is not local.

Part 2.1: Message-Passing-based k-hop Neighborhood Queries

After setting up the basic infrastructure, implement the k-hop neighbourhood query operation. This requires extending the network communication to support querying and aggregating features from neighbours up to k hops away.
Key Steps:

- $KHOP - NEIGHBORHOOD$ Operation: Extend the Worker class to include a method for k-hop neighbourhood queries, considering the δ parameters for limiting the number of messages at each hop.
- Aggregation and Message Passing: Implement logic for sending and receiving messages to/from neighbours to aggregate features according to the specified k hops and δ. parameters.
- Implement Logic: We collect every layer vertex's neighbours to the root vertex. For each layer, we have a set of neighbours to collect. We ask for the vertex feature and its neighbours, whose number is limited to the layer's index in δ. We add all collected vertex features and randomly select the required numbers of neighbours from all collected neighbours to be the new set of neighbours we need for the next layer.

Part 2.2: Marker-base k-hop Neighborhood Queries

Since the asynchronous implementation of the marker base is completely different from the message-passing-based implementation, we implemented a new k-hop code and ensured that the selection logic was the same.

The only difference from message-based is that we change all queries in the message-passing-based k-hop to wait for messages and markers to be actively delivered from neighbour vertexes at each layer. The delivered message includes the delivery path, which lets us know how to filter the message and search for the results K-hop needs.

Part 3: Message-Passing-based Synchronous "training"

Key Steps:

- Coordination of Epoch: To wait for each epoch to finish, we simply iterate from 1 to the final epoch and ask each worker to do the target epoch aggregation. We only start the next epoch when the previous one is finished.
- Neighborhood Aggregation in One Worker: If the target epoch's result is recorded, the worker simply sends out the result. Otherwise, the worker lets every vertex in its partition do the k-hop process and update its feature, which is executed in different threads to enhance performance.

Part 4: Message-Passing-based Asynchronous "training"

Key Steps:

- Shared Dict: We use a `epoch_dict` recording each vertex's epoch to keep causality across aggregation steps and epochs. In the k-hop process, a vertex must first check its neighbour epoch before doing the aggregation. It should exit if it finds a neighbour in a smaller epoch.
- History: For each vertex, we should keep a history of features for each epoch. Therefore, a vertex can ask for the feature corresponding to a certain epoch in the k-hop process.
- Neighborhood Aggregation: We start all epochs simultaneously. When a vertex wants to update its feature after k-hop, it should broadcast its update to the other 3 workers.

Part 5: Marker-based Asynchronous "training"

Because the data flow direction is opposite to the directed edge direction in the graph, the incoming and outgoing edges we discuss below are based on the data flow direction.
Key Steps:

- Initialization: Since the graph is a DAG, we simply find a vertex that doesn't have out edges as our initial vertex. By asking them to start a snapshot in each epoch, we are able to iterate every vertex in the graph. Upon receiving the snapshot message and epoch, the initial vertex records its current state and sends out its marker to all its out-edges.

- Marker: We use the marker inspired by both the Chandy-Lamport algorithm and the Epoch Snapshotting algorithm. Upon receiving a marker, the vertex records its current state and disables the edge the marker coming from. When there are no enabled in-edges, the vertex knows that it has received all required messages for updating itself by doing a k-hop operation. After that, it then sends the marker to all its out-edges. The marker isolates different epochs and keeps causality across aggregation steps and epochs.

- Message: A vertex sends out its feature in the format `v{id}f{feature}`. When a vertex receives a message from an enabled in-edge, it will prefix its id to the message and then keep sending it to all out-edges.

Difficulties

Part 2.1: Message-Passing-based k-hop Neighborhood Queries

- Partitioned Neighborhoods: Initially, we intended to implement k-hop through a distributed approach. That is to say, the root vertex would notify its δ_0 number of neighbours, these neighbours would then notify their δ_1 number of neighbours, and so on. After being notified at the final layer, they would return their vertex features. Subsequently, on each return from each layer, they would also return their vertex features and their neighbour vertexes and vertex features. This way, the collection could proceed layer by layer without requiring all the work to be done in a loop at the root vertex.

  However, this logic would lead to a problem, which is why we ultimately abandoned this method. Consider the following scenario: a vertex needs to execute k-hop with k=3 and $\delta = [2, 3, 2]$. However, when selecting all vertexes in the root vertex, it is possible to select a result as shown in the diagram. This does not meet the requirements of the k-hop algorithm.

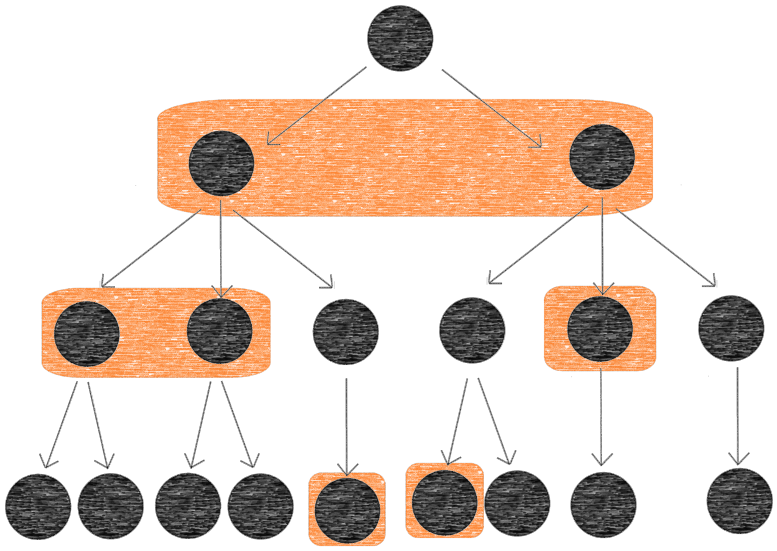

Figure 1: Wrong k-hop operation method

- Efficiency: We initially designed the current implementation that vertexes only communicate with their neighbours. This means that when collecting vertex features from the final layer, would require transmitting messages through each preceding layer back to the root vertex. However, we realized that this approach was resource-intensive. Therefore, we eliminated the indirect transmission and instead allowed the root vertex to directly communicate with the selected δ_n vertexes and request their features and neighbours.

Part 2.2: Marker-base k-hop Neighborhood Queries

- Partitioned Neighborhoods: To make the message passing of the Chandy Lamport algorithm and the marker be sent on the same channel, we still need to use the direction from the vertex with no outgoing edge to the direction of the vertex with no incoming edge. That is the transmission direction where problems may occur as described in 2.1. We solved this problem by adding the passed path to the message by prefixing the id of the vertex to the passed message.

- Efficiency: In this k-hop, we maintain the original transmission design. When the last layer receives the marker, the vertex characteristics are transmitted through each previous layer to transmit the message back to the vertexes that may need this information (k-hop requires this information, so each message should pass k layers after being sent).

Part 4: Message-Passing-based Asynchronous "training"

- A tricky bug occurred when we did the asynchronous training with parameters: epoch = 2, k = 1, and deltas = 5000. We found that some vertex features were different from the results of the synchronous training with the same parameters. After carefully debugging, we found that some vertexes asked for a vertex's feature even though that vertex hadn't been updated to the required epoch. The reason was that in the for loop where we check the epoch availability of the neighbours, we removed vertexes in the iterating list, which caused wrong indexes in iteration and let bad vertexes enter the next block.

Part 5: Marker-based Asynchronous "training"

Since we assume that the graph we are processing is a directed acyclic graph, we have not implemented the logic about processing cycles, but we have the following theory.

- Dealing with ring structures in epoch snapshotting & Chandy Lamport: We can detect the structure of the graph before starting to run. When a cycle is found, a notification is sent to each vertex on the cycle to inform them that they are within the cycle for subsequent operations. If there is no outgoing edge of the graph structure (incoming edge of the message) in the ring, we will add any point on the ring to the initialization vertex.

  When a vertex finds itself inside a ring, it compares the k-hop k to the number n of vertexes in the ring. The number of k-hops that the vertex can do will be the minimum value between k and n. That is, if the vertexes on the ring are not enough for the k-level k-hop, the k-hop of the vertex in the ring can only reach the n-level.

  The marker in the ring will be passed twice in the ring, which means that each vertex in the ring will get the marker from the same incoming edge of the message twice. This is different from the basic Chandy Lamport, but in this project, this will ensure that each vertex has enough features for k-hop.

Results

- Message-Passing-based results with RPC Protocol

Figure 2: RPC Async Epoch=2, K =1 and Delta=5000

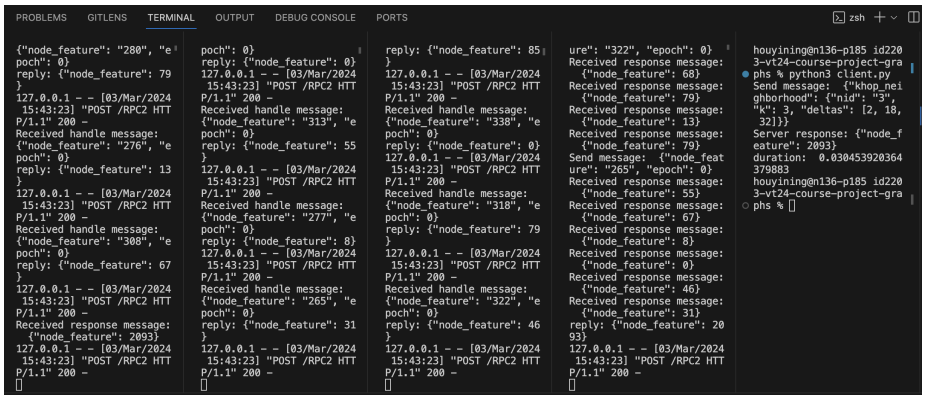

Figure 3: RPC K-hop

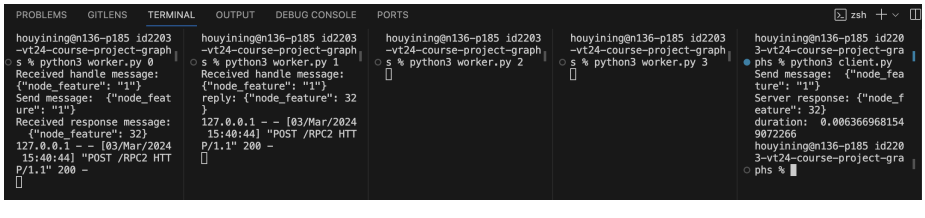

Figure 4: RPC vertex feature

- Message-Passing-based results with TCP Socket

Figure 5: TCP Socket Async Epoch=2, K =1 and Delta=5000

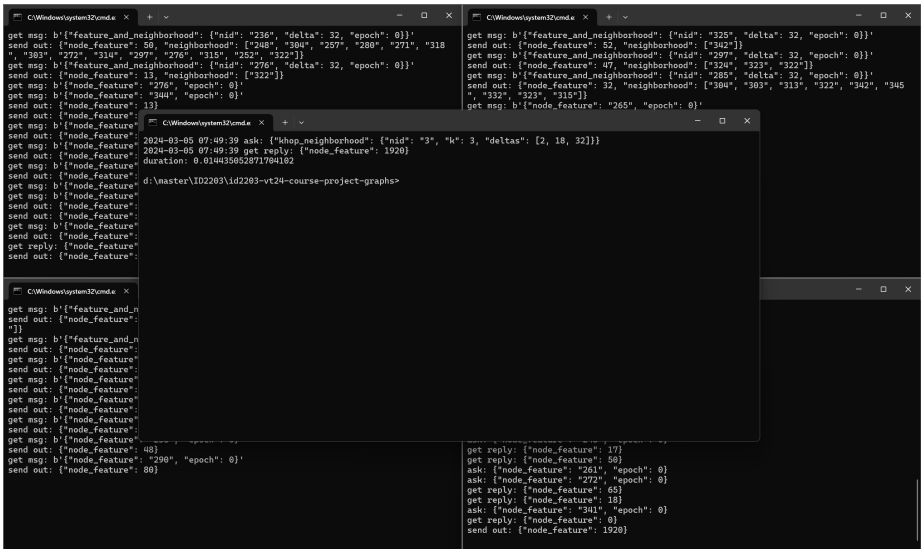

Figure 6: TCP Socket K-hop

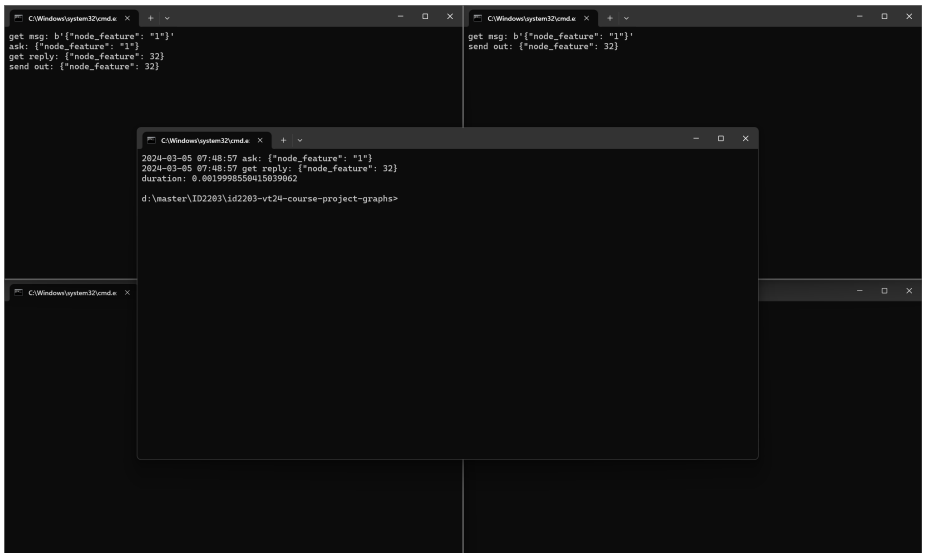

Figure 7: TCP Socket vertex feature

- Message-Passing-based results with Distributed Computing
  We started worker0 and worker2 on one machine, and worker1 and worker3 on the other machine.

Figure 8: RPC Async Worker0 2 Epoch=2, K =2 and Delta=[5000, 25000000]

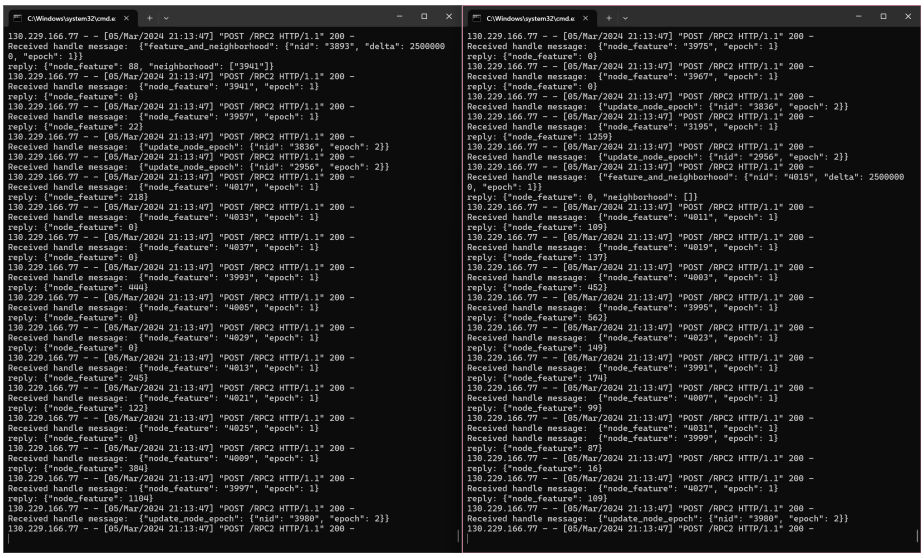

Figure 9: RPC Async Worker1 3 Epoch=2, K =2 and Delta=[5000, 25000000]

- Marker-based results with TCP Socket

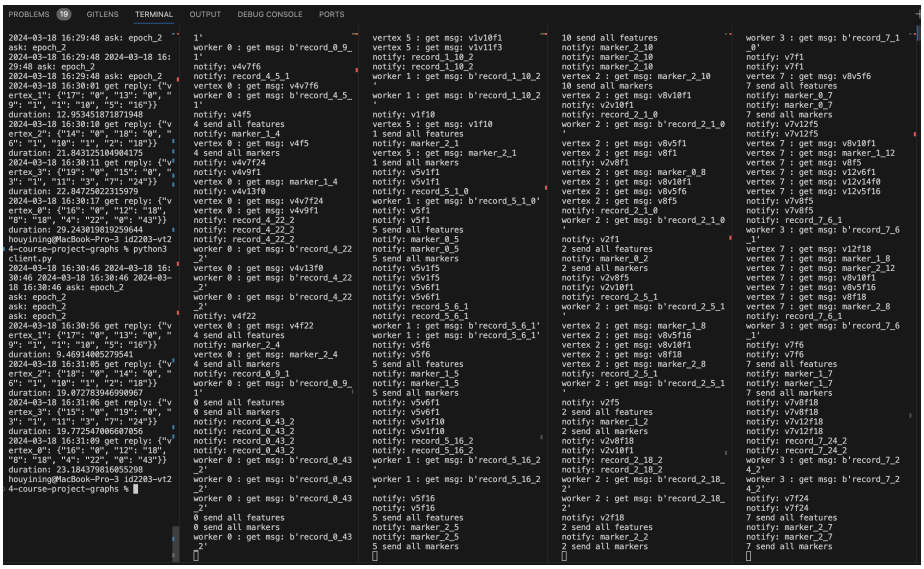

Figure 10: Marker Epoch=2, K =2 and Delta=[20, 400]

4 Testing

Test cases for Chandy-Lamport implementation:

Test cases :

- TC001: Verify vertex feature query result.
  - Preconditions/Postconditions: All servers are up and running
  - Inputs: ID of the vertex on the graph
  - Expected Outputs: Vertex feature value which stored in file `node_features.txt`
  - Test Steps: Call `query_node_feature(Node_ID)` method in `Client.py`
- Result:

```
hongjiang@MBPsomtlhorHong id2203-vt24-course-project-graphs % pyth
on3 client.py
Send message:  {"node_feature": "1"}
Server response: {"node_feature": 32}
took 0.008348226547241211 seconds to execute
```

Figure 11: Result of vertex feature query nid=1

- TC002: Verify k-hop neighborhood queries
  - Preconditions/Postconditions: All servers are up and running
  - Inputs: Vertex id, k number of jumps in the graph, and deltas array
  - Expected Outputs: sum of all vertexes feature value which starting from the current vertex with k jumps
  - Test Steps: Call `query_khop_neighborhood(Node_ID, k, deltas)` method in `Client.py`
- Result:

```
hongjiang@MBPsomtlhorHong id2203-vt24-course-project-graphs % pyth
on3 client.py
Send message:  {"khop_neighborhood": {"nid": "8", "k": 2, "deltas"
: [5000, 25000000]}}
Server response: {"node_feature": 368}
took 0.012660980224609375 seconds to execute
```

Figure 12: Result of k-hop neighborhood query nid=8, k=2 and deltas=[5000, 25000000]

- TC003: Verify synchronous training
  - Preconditions/Postconditions: All servers are up and running
  - Inputs: epochs number, k number of jumps in the graph, and deltas array size

  - Expected Outputs: return epoch diction which key is vertex id and value is the sum of all k jump paths feature value
  - Test Steps: Call `train_synchronize(epochs, k, deltas_size)` method in `Client.py`
  - Result: the same as we described above
  - Assertions: the returned result is the same as we trained and stored in file `check.py`

- TC004: Verify asynchronous training
  - Preconditions/Postconditions: All servers are up and running
  - Inputs: epochs number, k number of jumps in the graph, and deltas array size
  - Expected Outputs: return epoch diction which key is vertex id and value is the sum of all k jump paths feature value
  - Test Steps: Call `train_asynchronize(epochs, k, deltas_size)` method in `Client.py`
  - Result: the same as we described above
  - Assertions: the returned result is the same as we trained and stored in file `check.py`

- TC005: Verify asynchronous marker training
  - Preconditions/Postconditions: All servers are up and running
  - Inputs: epochs number(Here with k number of jumps in the graph, and deltas array size as constant variable in `worker_asy.py`)
  - Expected Outputs: return partition vertex from each worker with all small graph nodes path sum info
  - Test Steps: Call `train_asynchronize_marker(epochs)` method in `Client.py`
  - Result:

```
hongjiang@MBPsomtlhorHong id2203-vt24-course-project-graphs % python3 client.py
2024-03-21 21:37:11 ask: epoch_2
2024-03-21 21:37:11 ask: epoch_2
2024-03-21 21:37:11 2024-03-21 21:37:11 ask: epoch_2
ask: epoch_2
2024-03-21 21:37:25 get reply: {"vertex_1": {"13": "0", "17": "0", "9": "1", "1": "10", "5": "18"}}
duration: 14.224470853805542
2024-03-21 21:37:31 get reply: {"vertex_3": {"19": "0", "15": "0", "3": "1", "11": "3", "7": "39"}}
duration: 20.05662488937378
2024-03-21 21:37:31 get reply: {"vertex_2": {"18": "0", "14": "0", "6": "1", "10": "1", "2": "30"}}
duration: 20.568066120147705
2024-03-21 21:37:37 get reply: {"vertex_0": {"16": "0", "12": "26", "8": "26", "4": "42", "0": "88"}}
duration: 25.839859008789062
```

Figure 13: Result of train asynchronize marker epochs=2, K=3 and delta=20

  - Assertions: the returned result is the same as we trained and stored in file `check.py`

Accuracy

To check the accuracy of our training results, we provided two mock data files:

- neighbor.txt: Put all partitions together. Used to generate the whole graph and get all neighbours of one vertex.
- node_feature_dummy.txt: Assign 1 to all vertexes' features.

We also provided two test functions which check the accuracy of the results in depth(epoch) and breadth(deltas):

- test_mult_epochs: Use the dummy file and assign 2^{epoch} to default vertex feature. This function is used to test multiple epochs, k = 1, deltas = [1]. The result vertex feature should equal to 2^{epoch}.

Figure 14: Asy Dummy Epoch=5, K=1 and Delta=1

- test_all_neighbors: Use the dummy file and assign 1 to the default vertex feature. This function is used to test when every vertex collects all its neighbours in epoch = 1, k = 1, deltas = [5000]. We use 5000 because there are less than 5000 vertexes in the whole graph. The result vertex feature should be $1 + len(neighborsList)$.

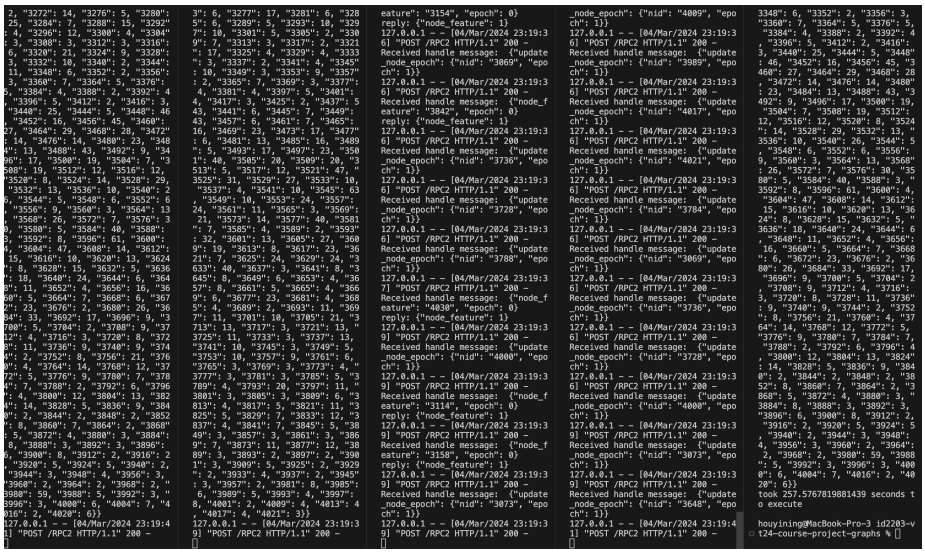

Figure 15: Asy Dummy Epoch=1, K=1 and Delta=5000

Time performance

Client

We used a decorator function `timeit` to calculate each request's time from send to receive. Add annotation `@timeit` to the required function.

```
houyining@MacBook-Pro-3 id2203-vt24-course-project-graphs % python3 client.py
Send message:  {"khop_neighborhood": {"nid": "3", "k": 3, "deltas": [2, 18, 32]}}
Server response: {"node_feature": 264}
took 0.017579078674316406 seconds to execute

houyining@MacBook-Pro-3 id2203-vt24-course-project-graphs %
```

Figure 16: Time Executing k-hop

Worker

We used the `line profiler` in Python, which provides detailed reports about the execution time of individual lines of Python code in a program. The Line Profiler breaks down the performance analysis to the line level, offering precise insights into where the most time is consumed.

To use the Line Profiler, we decorate the functions we wish to profile with a special annotation `@profile`, and then run the program with `kernprof`. The profiler generates a `.lprof` file and We wrote a script `parse_lprof_to_file.py`

to convert the file into a report that includes the number of times each line was executed and the time each line took to execute. This granular level of detail helps us optimize our code by focusing on the lines that have the most significant impact on performance.

```
Total time: 17834 s
File: worker.py
Function: send_message at line 231

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   231                                           @profile
   232                                           def send_message(self, node, message):
   233     40700  364319680.0   8951.3      2.0      print("Send message: ", message)
   234     41223      15713.0      0.4      0.0      while True:
   235     41223       6589.0      0.2      0.0          try:
   236     41223      43080.0      1.0      0.0              port = 12345 + int(node) % NUM_PARTITIONS
   237     41223     551541.0     13.4      0.0              proxy = xmlrpc.client.ServerProxy(f"http://localhost:{port}")
   238     41223        2e+10 413537.6     95.6              response = proxy.handle_msg(message)
   239     40700  421759362.0  10362.6      2.4              print("Received response message: ", response)
   240     40700      14935.0      0.4      0.0              return response
   241       523        386.0      0.7      0.0          except Exception as e:
   242                                                       # print(e)
   243                                                       # print("!!!!!!RPC exception!!!!!!, retrying...")
   244       523       2942.0      5.6      0.0              continue
```

Figure 17: Line Inspection of send_message function

Space performance

We used the `memory profiler` in Python, which provides detailed insights into the memory consumption of individual lines of code, making it easier to identify memory-intensive parts of the application.

To use the Memory Profiler, we run the program with `mprof` and it generates `.dat` files recording line-by-line memory usage of each function. To make the result more visual, we can use `mprof plot` to generate a graph.

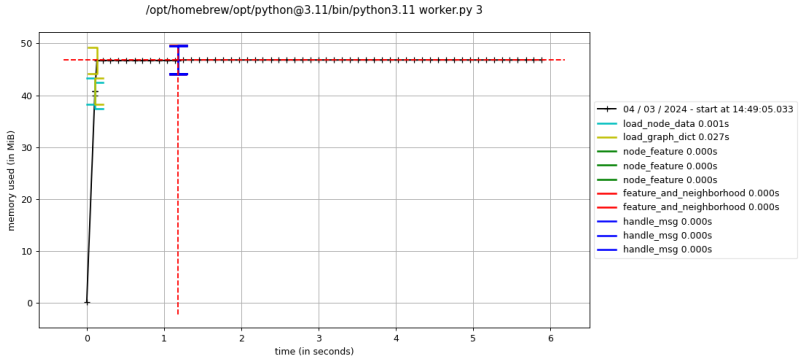

Figure 18: Memory usage in khop

However, if the `.dat` file becomes too big (e.g. train), it's useless to convert it

into a graph like the above. Therefore, we wrote a script `parse_dat_to_graph.py` to show the general memory usage of each function.

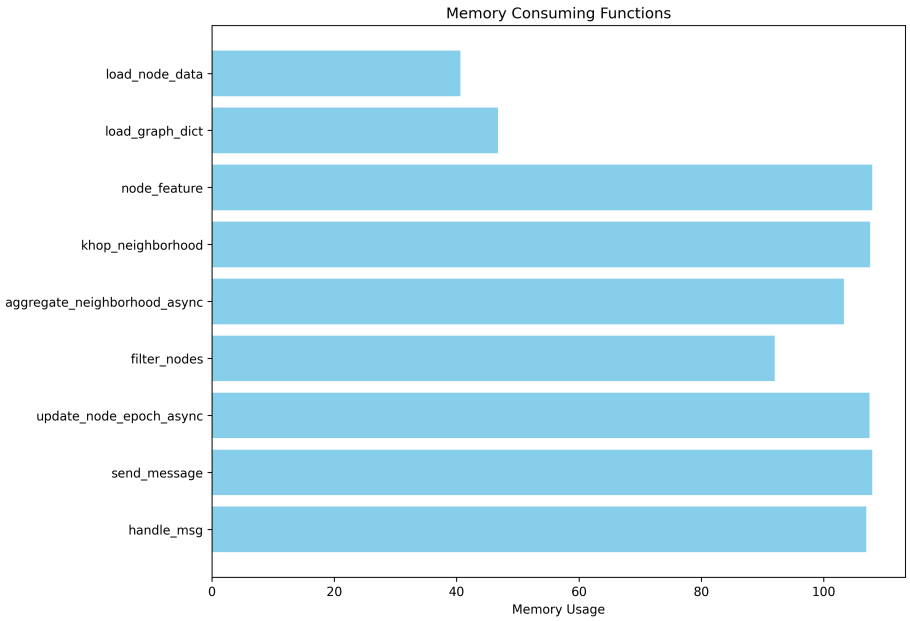

Figure 19: Memory usage in train

Comparison

Message-passing-based RPC Vs Socket

To compare the performance between the socket solution and the RPC solution, we chose a list of parameters: epoch = 2, k = 1, deltas = 5000. We ran both the synchronous train and asynchronous train, and compared the time it took.

For the RPC solution, it took 510s to finish the asynchronous train and 515s to finish the synchronous train.

Figure 20: RPC Asy Train Time

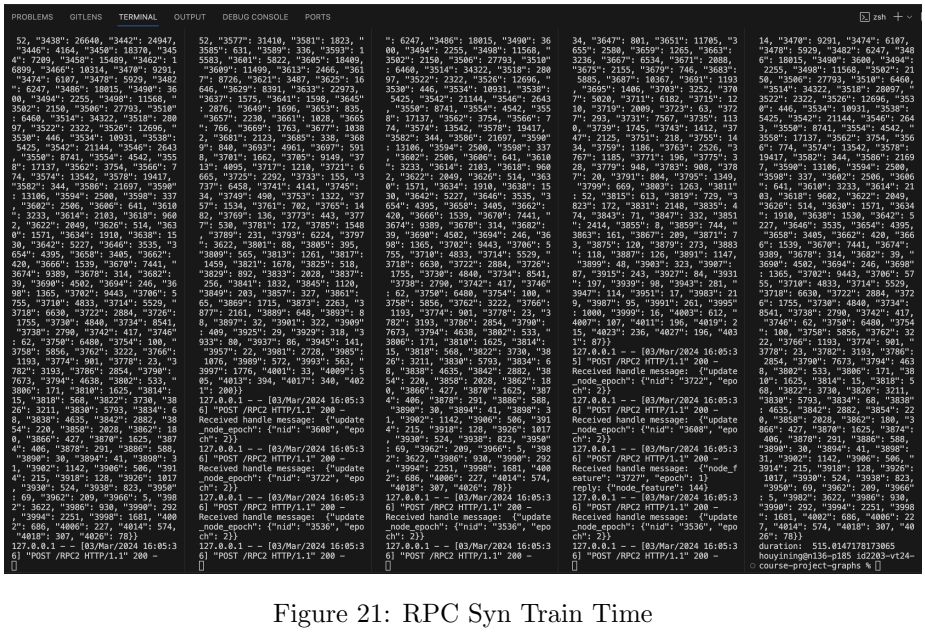

Figure 21: RPC Syn Train Time

For the socket solution, it took 220s to finish the asynchronous train and 241s to finish the synchronous train.

Figure 22: Socket Asy Train Time

Figure 23: Socket Syn Train Time

We compared the train time between the RPC and Socket solution. As shown in the following tables, Socket is faster than RPC in most cases. The reasons might be:

1. Lower Level of Abstraction:

Socket operates at the transport layer of the network stack. It provides a closer-to-hardware way of communication, reducing the level of abstraction and thus, often resulting in higher performance and less overhead. We manage data transfers directly, with no extra processing or encapsulation steps.
RPC abstracts the details of network communication, often using higher-level protocols (like HTTP, gRPC) for data transmission and involves serializing and deserializing data during the communication process. RPC operates at the application layer, providing a higher level of abstraction compared to sockets.

2. Serialization and Deserialization: During RPC communication, data often needs to be serialized and deserialized. This process consumes time and computational resources, especially for complex data structures.

3. Protocol Overhead: RPC frameworks usually employ more complex protocols, such as HTTP/HTTPS or gRPC. These protocols introduce more overhead than raw TCP or UDP sockets due to additional headers, and state management.

epoch, k, deltas	1, 1, [5000]	1, 2, [5000, 25000000]	2, 1, [5000]	2, 2, [5000, 25000000]
RPC	153	727	515	1393
Socket	14	83	30	169
Distributed RPC	166	940	307	2462

Table 1: Synchronous Train Time RPC vs Socket

epoch, k, deltas	1, 1, [5000]	1, 2, [5000, 25000000]	2, 1, [5000]	2, 2, [5000, 25000000]
RPC	153	689	510	1391
Socket	15	80	29	170
Distributed RPC	140	844	298	2339

Table 2: Asynchronous Train Time RPC vs Socket

We also conducted RPC solutions on distributed machines. We found that when k increases, Distributed RPC is much slower than local RPC. The unstable network may cause a lower speed since k-hop needs more interactions among different machines.

Socket Marker-based Vs Message-passing-based

- Neighborhood query execution time:
  We pick 4 groups of parameters to test the training speed between two

solutions. One epoch and one layer. Then either increase the epoch or layer number. Lastly, increase both epoch and layer number. Message-passing-based training is much faster than marker-based training.
In message-based training, every vertex starts their k-hop simultaneously, and only epoch can limit their speed to train. In the marker-based training, the message needs to be passed from bottom to top. Only when the bottom layer vertex finishes its training, can the topper layer vertex start its training. It's still asynchronous because the bottom layer vertex and top layer vertex can be in different epochs.

epoch, k, deltas	1, 1, [20]	1, 2, [20, 400]	2, 1, [20]	2, 2, [20, 400]
Message-passing	0.015	0.019	0.53	0.54
Marker	17	19	24	26

Table 3: Train Time Message-passing vs Marker

- Read throughput: We pick 2, 2, [20, 400] to count the number of messages for each solution.
  For Message-passing asynchronous training, we 'Ask' a vertex about its information and receive its reply. 'Tell' is a kind of 'Ask' which doesn't need a reply. 'Receive' and 'Reply' are messages received by the vertex and the reply is sent back.
  The number of messages sent(Ask + Tell) is equal to the number of messages received(Receive): 97+78 = 175+1(1 more received from the client). And the number of messages asked(Ask) is equal to the number of messages replied(Reply): $97 = 97 + 1$(1 more reply to client).
  For Marker-based asynchronous training, we only have a one-way message pass. The number of messages sent(Tell) is equal to the number of messages received(Receive): $210 = 210 + 74$(74 more Tell is the messages recorded).
  The messages passed in the message-passing solution are slightly more than the messages in the marker-based solution. That's because, in the message-passing solution, most messages are passed in a round trip compared with a one-way pass in a marker-based solution.

Messages	Ask	Ask receive	Tell	Receive	Reply	All
Message-passing	97	97	78	176	98	546
Marker	0	0	284	210	0	494

Table 4: Messages Message-passing vs Marker

- Socket utilized: We also pick 2, 2, [20, 400] to count the number of sockets created for each solution.
  In the message-passing solution, whenever we Ask(97) or Tell(78) a message, we create a socket to handle the communication. When a vertex Receives (176) a message, it creates a new client socket to handle the message. Besides, we have 4 workers who keep listening on 4 server sockets. Therefore, $97 + 78 + 176 + 4 = 355$ sockets are utilized.
  In a market-based solution, similar to the above, we create a socket when we Tell(284) a message and Receive(210) a message. For each vertex, we start a socket to listen on a server socket(20). Therefore, $284 + 210 + 20 = 514$ sockets are utilized.
  The sockets we used for the marker-based solution are much more than the sockets created in the message-passing solution. The main reason is that for each vertex we create a socket, and when the number of vertex increases, it will impose heavy stress on the hardware. Besides, we use round trip in the message-passing solution, so most messages can be handled within the same sockets.

Socket	Number
Message-passing	355
Marker	514

Table 5: Socket Message-passing vs Marker

- Latency for GET operations:
  Since we use Socket to pass messages for both message-passing training and marker-based training, the only difference is the message type. We use `json` to pass the message in the message-passing solution, while `string` in the marker-based solution. However, there is no obvious difference in the GET latency for these two types.

Latency	Client send → Server receive	Server send → Client receive
Message-passing	0.0001	0.00003
Marker	0.0001	0.00003

Table 6: Latency Message-passing vs Marker

- Memory utilization:
  There is no obvious difference in total memory use for two approaches. For the message-passing solution, there is no fluctuation during the whole training process. However, in the marker-based approach, we can see that the memory utilization first peaked and then decreased, floating at a lower level, which is consistent with its training convergence.

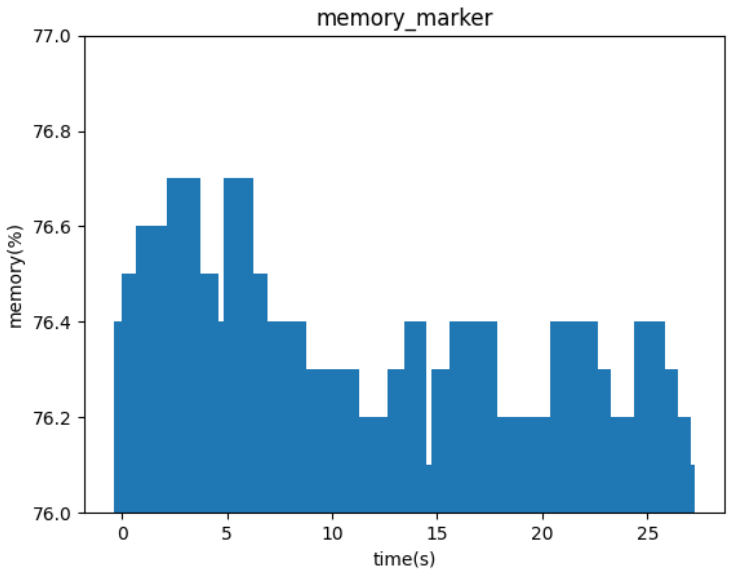

Figure 24: Marker Memory utilization

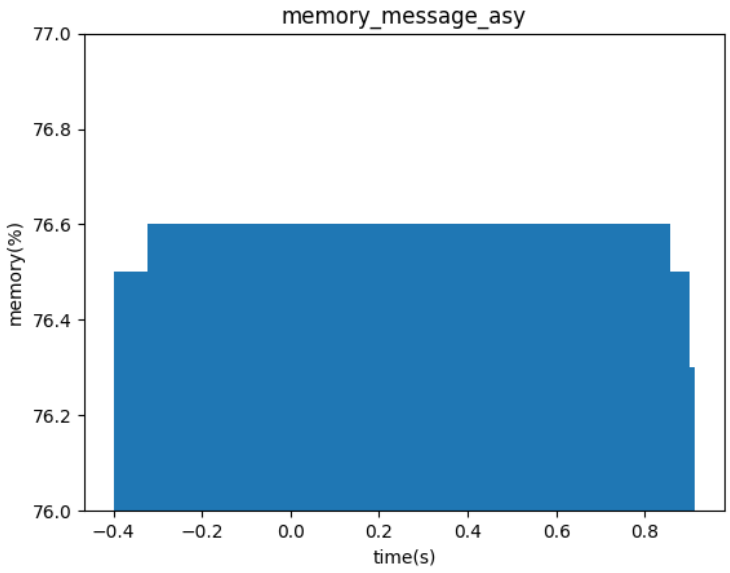

Figure 25: Message Memory utilization

5 Future Work

For epoch synchronization, we have achieved the best effort broadcast and implemented failure re-transmission. Subsequent implementations could utilize reliable broadcast or uniform reliable broadcast. This may optimize performance when running workers on different computers.
Since too many sockets cannot be opened at the same time, we cannot run the asynchronous implementation of our marker base in Facebook's original directed acyclic graph.
Due to Python code being interpreted line by line, while C/C++ is compiled directly into machine code, C/C++ generally offers higher execution efficiency. Therefore, if we can implement our algorithms in C/C++ in the future, we will definitely achieve higher efficiency, which will also be part of our future work.

6 Summary

6.1 What we achieved

We have implemented a distributed graph store system aimed at managing graph-based data across multiple workers. Here's a brief summary of the implemented features:

1. Basic Infrastructure: We designed a distributed system where graph partitions are distributed over 4 workers. Each worker runs as a separate process, supporting basic `Get(nid)` operations to retrieve vertex features by their ID.

2. Neighborhood Queries: We implemented `Khop_Neighborhood(nid, k, deltas)` operations to support k-hop neighbourhood queries, which calculate the aggregated features of a vertex's neighbourhood up to k hops away.

3. Message-Passing-Based Neighborhood Aggregation: To simulate the neighbourhood aggregation process of GNN training, we introduced a `Train(epochs)` operation. This iterates over vertexes to compute new weights based on k-hop neighbourhood aggregations, with each vertex storing a history of aggregation weights. `Future` mechanism was considered to manage the completion of epochs and support synchronous training operations. We used two protocols: RPC and Socket, each with similar logic in both asynchronous training and synchronous training.

4. Marker-Based Asynchronous Training: We combined the Chandy-Lamport algorithm with the Epoch Snapshotting algorithm. We sent `snapshot` to a certain initial vertex to initialize a snapshot(epoch). Each snapshot has a marker, when a vertex receives the marker, it records its state. After receiving all markers, the vertex is ready for iterative k-hop neighbourhood aggregations. The marker and messages kept causality across aggregation steps and epochs through

a FIFO channel, which is implemented by TCP Socket.

5. Performance Evaluation: For Message-Pasing-Based training, we provided and compared two versions of solutions based on Socket and RPC. To compare the message-based training approach to the marker-based training method, we used the same message-passing protocol: TCP Socket. The key metrics we applied include training execution time, which will provide the efficiency of each method; the number of messages, which could help understand the throughput capabilities; and the number of sockets utilized, as this might affect scalability and resource usage.

6.2 Existing problem

All our tests were conducted on the same computer, meaning all 4 workers were running on a single machine. We attempted distributing our workers across different computers, but due to network instability, there were fluctuations in message transmission and reception. This resulted in slowdowns during message exchange or invocation processes, thus affecting the overall effectiveness.