Brandon Cheng, chen7381
Shengya Zhang, zhan9051

1. Introduction

This project implements a distributed system for training a machine learning model using multiple compute nodes, managed by a central coordinator. The goal is to enable distributed training of a multilayer perceptron (MLP) by leveraging multiple nodes for parallel computation. This system follows a parameter server model where a central model is updated using gradients computed by worker nodes.

2. Design

The system consists of three main components:

Coordinator Node:

- Manages model training and orchestrates communication between compute nodes.
- Sends initial model weights to compute nodes.
- Aggregates gradients from compute nodes and updates the global model.

Compute Nodes:

- Receive weights and training data from the coordinator.
- Independently train on assigned data and compute weight gradients.
- Return computed gradients to the coordinator for aggregation.

Client:

- Initiates training jobs by sending job specifications to the coordinator.

3. Implementation Details

Coordinator Node:

The coordinator has one function - train().

The **train()** function initializes an instance of the mlp model (almighty) and loads a random training file to get global weights v and w setup.
Then, it identifies available compute nodes by reading from ‘compute_nodes.txt’ and tests each entry in the text file to determine a list of online compute nodes.

The coordinator initializes a global shared gradient to store aggregated weight updates, and also uses a mutex lock to ensure thread safety on the shared gradient weights.

During each round of training, the main thread populates the training file queue that is to be worked through by compute nodes, which is also guarded by a mutex lock for concurrency control. Then the worker threads will pop tasks from the queue and work through them, while the main thread will be blocked waiting on the conditional variable tied to bool round_complete. Once a worker thread notices it has processed the last file, it will switch round_complete to true, and notify the main thread. The main thread will aggregate the results of the shared gradients, then move on to the next round and the repeat the same process. When all rounds of training is completed, the workers are notified through the file_cv conditional variable, then they will check the the done_training variable and exit the thread.

Finally, the validation error will be returned to the client.

Compute Nodes:

Compute nodes have two functions, check_availability() and train().

check_availability() returns a boolean value indicating if the node passed the probability checking. True means node is available, and vice versa. This is then used to determine if the node should sleep in random scheduling or reject a task in load balancing.

train() is where the local model training happens on each node. It is given the necessary arguments like training filename, global v and w weights, and other parameters to train the model. It is also given a boolean value indicating whether it should sleep or not from the result of check_availability(). It returns the calculated gradients after training.

Client:

The client only has a main function that contacts the coordinator using the coordinator's IP and port from the command line arguments. Dir_path, rounds, and epochs are also passed in from the command line for training. However, h=24, k=26, and eta=0.0001 are hard coded in the client because these parameters are not changed often as they affect the error rate.

4. Testing and Performance Evaluation

In this project, we used 6 nodes, below is the information and the port numbers;

DistributedMLP > compute_nodes.txt

```
1   localhost, 9091
2   localhost, 9092
3   localhost, 9093
4   localhost, 9094
5   localhost, 9095
6   localhost, 9096
```

In all test cases, h, k, and eta are not changed as previously defined in section 3.

Test Cases

- Scenario 1: All compute nodes have the same low(0.2) or high(0.6) load probability.

  Using all 6 compute nodes with 0.2 probability; 20 rounds; 15 epochs; random scheduling

  Time: 57.166s

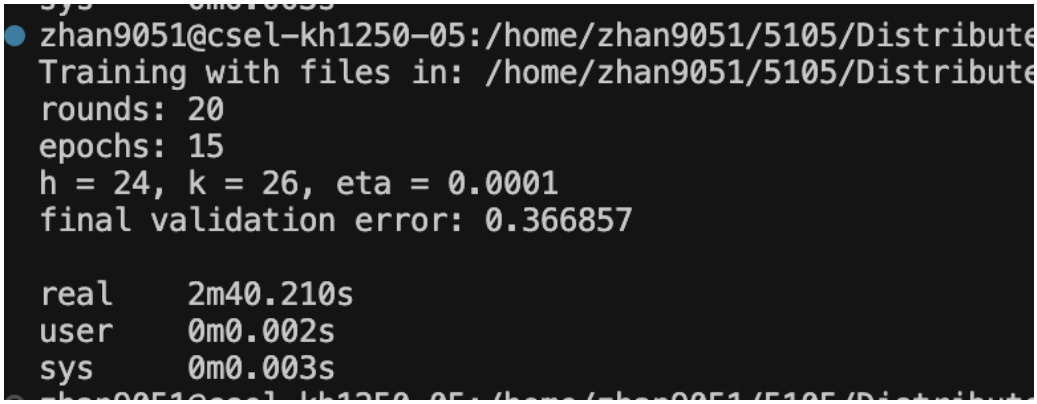
```
zhan9051@csel-kh1250-05:/home/zhan9051/5105/DistributedMLP/build $ time ./client localhost 9099 /ho
me/zhan9051/5105/DistributedMLP/letters 20 15
Training with files in: /home/zhan9051/5105/DistributedMLP/letters
rounds: 20
epochs: 15
h = 24, k = 26, eta = 0.0001
final validation error: 0.366857

real    0m57.166s
user    0m0.002s
sys     0m0.003s
zhan9051@csel-kh1250-05:/home/zhan9051/5105/DistributedMLP/build $
```

  Final validation error:0.366857

  Using all 6 compute nodes with 0.6 probability; 20 rounds; 15 epochs ;random scheduling;

  Time: 2m40.21s



Final validation error:0.366857

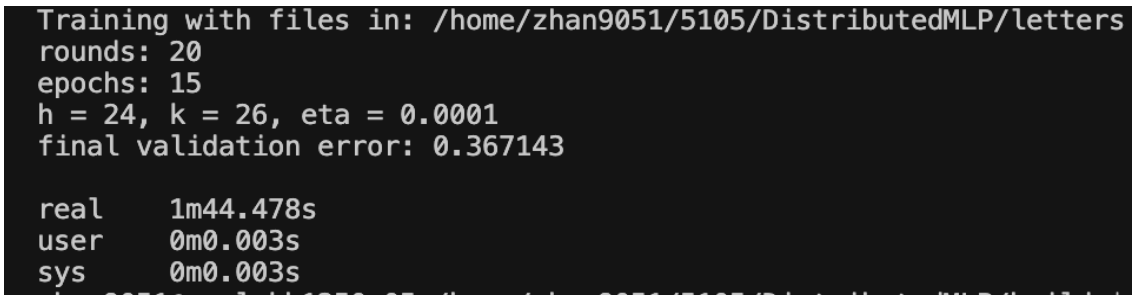
- Scenario 2: Compute nodes have varying load probabilities.

  Using 6 compute nodes with 6 different load probability：

  0.1 ；0.2；0.3；0.4；0.5；0.6 probability;

  20 rounds; 15 epochs ;random scheduling;

  Time:1m44.478s



Final validation error:0.367143

Using 6 compute nodes with 6 different load probability:

0.1 ；0.2；0.3；0.4；0.5；0.6 probability;

20 rounds; 15 epochs; load-balancing scheduling;

All the previous compute nodes settings are the same as the settings that we used in random scheduling, but on the coordinator part, use ‘2’ which means load-balanced scheduling.

Time: 41.058s

```
Training with files in: /home/zhan9051/5105/DistributedMLP/letters
rounds: 20
epochs: 15
h = 24, k = 26, eta = 0.0001
final validation error: 0.366857

real    0m41.058s
user    0m0.001s
sys     0m0.004s
```

Final validation error:0.366857

- Scenario 3: Comparison of random vs. load-balanced scheduling.

  Using all 6 compute nodes with 0.2 probability; 20 rounds; 15 epochs ;random scheduling;

  Time: 1m45.146s

```
chen7381@csel-remote-lnx-02:/home/chen7381/5105/DistributedMLP/build $ time ./client localhos
t 9000 ../letters 20 15
Training with files in: ../letters
rounds: 20
epochs: 15
h = 24, k = 26, eta = 0.0001
final validation error: 0.366857

real    1m45.146s
user    0m0.005s
sys     0m0.003s
```

  Final validation error:0.366857

  Using all 6 compute nodes with 0.2 probability; 20 rounds; 15 epochs ;load_balancing scheduling;

  Time: 1m11.624s

```
chen7381@csel-remote-lnx-02:/home/chen7381/5105/DistributedMLP/build $ time ./client localhos
t 9000 ../letters 20 15
Training with files in: ../letters
rounds: 20
epochs: 15
h = 24, k = 26, eta = 0.0001
final validation error: 0.367143

real    1m11.624s
user    0m0.007s
sys     0m0.007s
```

Final validation error:0.367143

5. Performance Metrics

time	0.2	0.6
Random	57.166s	160.21s
load-balancing	37.676s	79.073s

The above metric shows the comparison of random vs. load-balanced scheduling with the same low(0.2) or high(0.6) load probability.

6. Performance Analysis

When using random scheduling, execution time significantly increases as the load probability increases. This suggests that random scheduling does not efficiently distribute the workload, causing high variance in execution time depending on the node availability.

With varying load probability, the execution time remains lower, which means the varying load probability scheduling is more efficient in handling workload distribution, reducing execution time.

The error rate remains nearly constant across different configurations, around **0.366857 - 0.367143**, indicating that scheduling policies mainly influence execution time rather than model accuracy.

This project successfully implements a distributed MLP training system. The system effectively distributes training tasks across compute nodes, with the coordinator aggregating updates. Load balancing improves resource utilization. Future improvements could include dynamic load adjustment and fault tolerance mechanisms.