COMP SCI 744 - Assignment 2

GROUP 14

Suyash Raj
Handan Hu
Amr Alazali

Introduction

In this assignment, we set up the environment on Cloudlab and created a cluster of 4 VMs.

We utilized PyTorch to train a VGG-11 network on the CIFAR-10 dataset, and performed distributed data parallel training with collective communication frameworks like Gloo.

- We set the random seed to 14 in all tasks so that we can start from the same model and compare the result.
- We ran the training for 1 epoch with a batch size of 256, and printed the loss value every 20 iterations.
- We recorded the average run time for the first 40 iterations, excluding the first one.

Part 1

In this part, we trained the model on a single machine (node0)

```
(base) root@node0:/users/sneo/assignment2/part1# python main.py
Epoch 1 [0/196] - Loss: 2.5538
Epoch 1 [20/196] - Loss: 5.8252
Average time per iteration after 40 iterations: 2.5220 sec
Epoch 1 [40/196] - Loss: 4.2190
Epoch 1 [60/196] - Loss: 3.6066
Epoch 1 [80/196] - Loss: 3.2830
Epoch 1 [100/196] - Loss: 3.0839
Epoch 1 [120/196] - Loss: 2.9496
Epoch 1 [140/196] - Loss: 2.8516
Epoch 1 [160/196] - Loss: 2.7745
Epoch 1 [180/196] - Loss: 2.7090
Test set: Average loss: 2.2750, Accuracy: 2033/10000 (20%)

Epoch 1 complete time: 517.1583 sec
```

Observations

- Average loss: **2.2750**
- Accuracy: **20%**
- Average time per iteration (for the first 40 iterations): **2.5220 sec**
- Completion time for one epoch: **517 sec**

Part 2a

In this part we implemented Distributed Data Parallel Training with gather and scatter communication collectives. Our algorithm manually implements gradient synchronization in a data-parallel distributed training setup with 4 workers, where the data is equally distributed across them. The process works as follows:

1. Each worker independently computes gradients on its local data partition.
2. Gradients are gathered on rank 0 using `dist.gather()`, collecting contributions from all 4 workers.
3. On rank 0, the gradients are averaged across all workers.
4. The averaged gradient is scattered back to all workers using `dist.scatter()`.
5. Each worker replaces its local gradient with the globally averaged gradient before performing parameter updates.

```
for param in model.parameters():
    if param.grad is not None:
        grad = param.grad
        if rank == 0:
            grad_list = [torch.zeros_like(grad) for _ in range(world_size)]
        else:
            grad_list = None
        dist.gather(grad, gather_list=grad_list, dst=0)
        if rank == 0:
            grad_avg = grad_avg = torch.stack(grad_list, dim=0).mean(dim=0)
            scatter_list = [grad_avg.clone() for _ in range(world_size)]
        else:
            scatter_list = None
        aggregated_grad = torch.zeros_like(grad)
        dist.scatter(aggregated_grad, scatter_list=scatter_list, src=0)
        param.grad = aggregated_grad
# Adjust learning weights
optimizer.step()
```

Below are the training and test losses of each of our 4 workers trained using our manual implementation of Distributed Data Parallel Training.

```
Rank 0, Epoch 0, Batch 0, Bacth Size: 64, Loss: 2.4944
Rank 0, Epoch 0, Batch 20, Bacth Size: 64, Loss: 6.0714
Average time per iteration after 40 iterations: 1.3634 sec
Rank 0, Epoch 0, Batch 40, Bacth Size: 64, Loss: 4.5746
Rank 0, Epoch 0, Batch 60, Bacth Size: 64, Loss: 3.8551
Rank 0, Epoch 0, Batch 80, Bacth Size: 64, Loss: 3.4674
Rank 0, Epoch 0, Batch 100, Bacth Size: 64, Loss: 3.2278
Rank 0, Epoch 0, Batch 120, Bacth Size: 64, Loss: 3.0585
Rank 0, Epoch 0, Batch 140, Bacth Size: 64, Loss: 2.9346
Rank 0, Epoch 0, Batch 160, Bacth Size: 64, Loss: 2.8339
Rank 0, Epoch 0, Batch 180, Bacth Size: 64, Loss: 2.7520
Test set: Average loss: 2.0295, Accuracy: 2009/10000 (20%)
```

```
Rank 1, Epoch 0, Batch 0, Bacth Size: 64, Loss: 2.3879
Rank 1, Epoch 0, Batch 20, Bacth Size: 64, Loss: 7.4142
Average time per iteration after 40 iterations: 1.3631 sec
Rank 1, Epoch 0, Batch 40, Bacth Size: 64, Loss: 5.3059
Rank 1, Epoch 0, Batch 60, Bacth Size: 64, Loss: 4.3522
Rank 1, Epoch 0, Batch 80, Bacth Size: 64, Loss: 3.8375
Rank 1, Epoch 0, Batch 100, Bacth Size: 64, Loss: 3.5249
Rank 1, Epoch 0, Batch 120, Bacth Size: 64, Loss: 3.3108
Rank 1, Epoch 0, Batch 140, Bacth Size: 64, Loss: 3.1508
Rank 1, Epoch 0, Batch 160, Bacth Size: 64, Loss: 3.0253
Rank 1, Epoch 0, Batch 180, Bacth Size: 64, Loss: 2.9220
Test set: Average loss: 2.0223, Accuracy: 2032/10000 (20%)
```

```
Rank 2, Epoch 0, Batch 20, Bacth Size: 64, Loss: 6.5694
Average time per iteration after 40 iterations: 1.3632 sec
Rank 2, Epoch 0, Batch 40, Bacth Size: 64, Loss: 4.8664
Rank 2, Epoch 0, Batch 60, Bacth Size: 64, Loss: 4.0360
Rank 2, Epoch 0, Batch 80, Bacth Size: 64, Loss: 3.5987
Rank 2, Epoch 0, Batch 100, Bacth Size: 64, Loss: 3.3317
Rank 2, Epoch 0, Batch 120, Bacth Size: 64, Loss: 3.1484
Rank 2, Epoch 0, Batch 140, Bacth Size: 64, Loss: 3.0143
Rank 2, Epoch 0, Batch 160, Bacth Size: 64, Loss: 2.9077
Rank 2, Epoch 0, Batch 180, Bacth Size: 64, Loss: 2.8152
Test set: Average loss: 2.0235, Accuracy: 2019/10000 (20%)
```

```
Rank 3, Epoch 0, Batch 0, Bacth Size: 64, Loss: 2.5215
Rank 3, Epoch 0, Batch 20, Bacth Size: 64, Loss: 6.0958
Average time per iteration after 40 iterations: 1.3632 sec
Rank 3, Epoch 0, Batch 40, Bacth Size: 64, Loss: 4.5389
Rank 3, Epoch 0, Batch 60, Bacth Size: 64, Loss: 3.8422
Rank 3, Epoch 0, Batch 80, Bacth Size: 64, Loss: 3.4551
Rank 3, Epoch 0, Batch 100, Bacth Size: 64, Loss: 3.2221
Rank 3, Epoch 0, Batch 120, Bacth Size: 64, Loss: 3.0559
Rank 3, Epoch 0, Batch 140, Bacth Size: 64, Loss: 2.9312
Rank 3, Epoch 0, Batch 160, Bacth Size: 64, Loss: 2.8308
Rank 3, Epoch 0, Batch 180, Bacth Size: 64, Loss: 2.7437
Test set: Average loss: 2.0252, Accuracy: 2022/10000 (20%)
```

Observations

- Average loss: **2.0295**
- Accuracy: **20%**
- Average time per iteration (for the first 40 iterations): **1.3857 sec**

Discussion

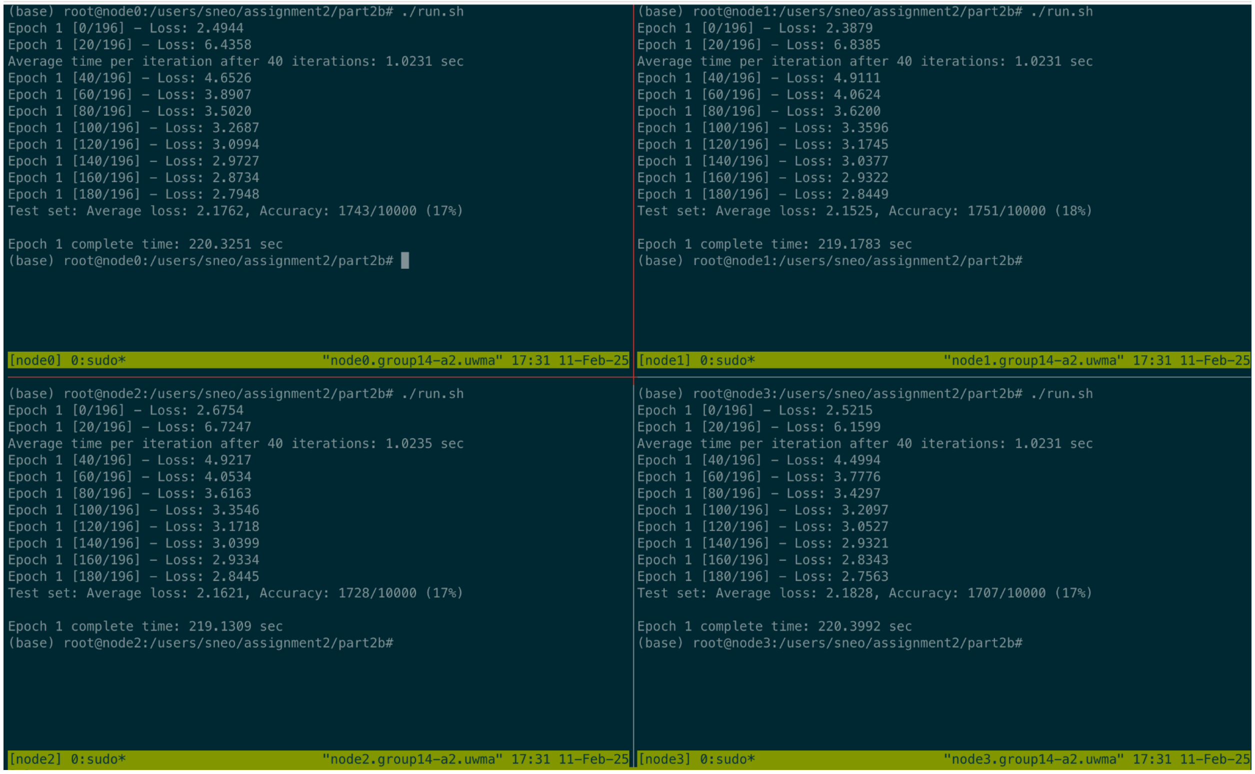
The average time per iteration is shorter compared to Part 1 but longer than AllReduce implementation of DDP in part2b. Training is faster when compared to non-parallel training in part 1 because each worker processes a quarter of the data simultaneously as opposed to having a single work compute gradients of all the data in a batch. This reduces the overall training time by leveraging multiple computing units simultaneously, improving throughput and reducing bottlenecks. The reason the Allreduce implementation is faster is because it doesn't have the additional overhead required to gather gradients on node 0 and then scatter them to all other nodes. Since node 0 performs the gradient aggregation and scattering it becomes a bottleneck which will become more noticeable as we increase the number of nodes.

Part 2b

In this part, we implemented gradient sync using allreduce collective. We got the average on each node by dividing with the number of workers and used SUM operation:

```
# sync gradient with allreduce
for param in model.parameters():
    param.grad /= 4
    dist.all_reduce(param.grad, op=dist.ReduceOp.SUM, group=group, async_op=False)
```

We trained the model on 4 workers, and here we recorded the result of node0:



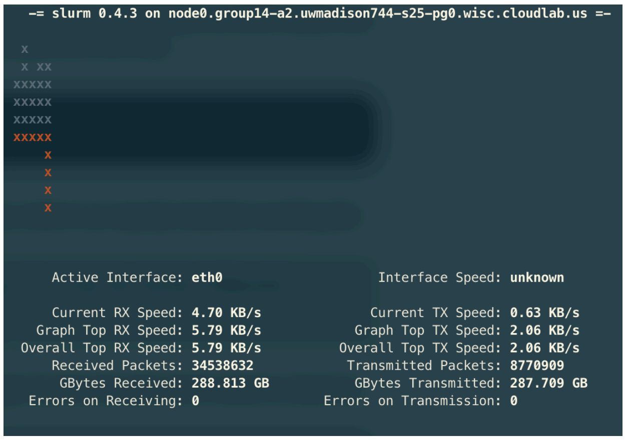
Observations

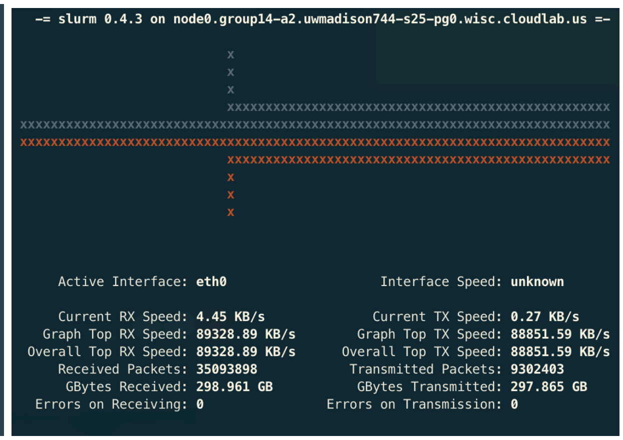
- Average loss: **2.1762**
- Accuracy: **17%**
- Average time per iteration (for the first 40 iterations): **1.0231 sec**
- Completion time for one epoch: **220 sec**

The average time per iteration is shorter compared to Part 1 and Part 2a, which suggests that gradient synchronization with AllReduce has better performance than gather and scatter. This is because the AllReduce algorithm sums gradients in a ring topology, where each worker shares its gradient with neighboring nodes. This mechanism eliminates the centralized bottleneck when node0 collects and processes all gradients, therefore reducing communication overhead.

This improvement can be further demonstrated by inspecting the network profile, as the amount of data received and transmitted is lower than that in Part 2a:

- Data Received (node 0) = 10.15 GB
- Data Transmitted (node0) = 10.16 GB

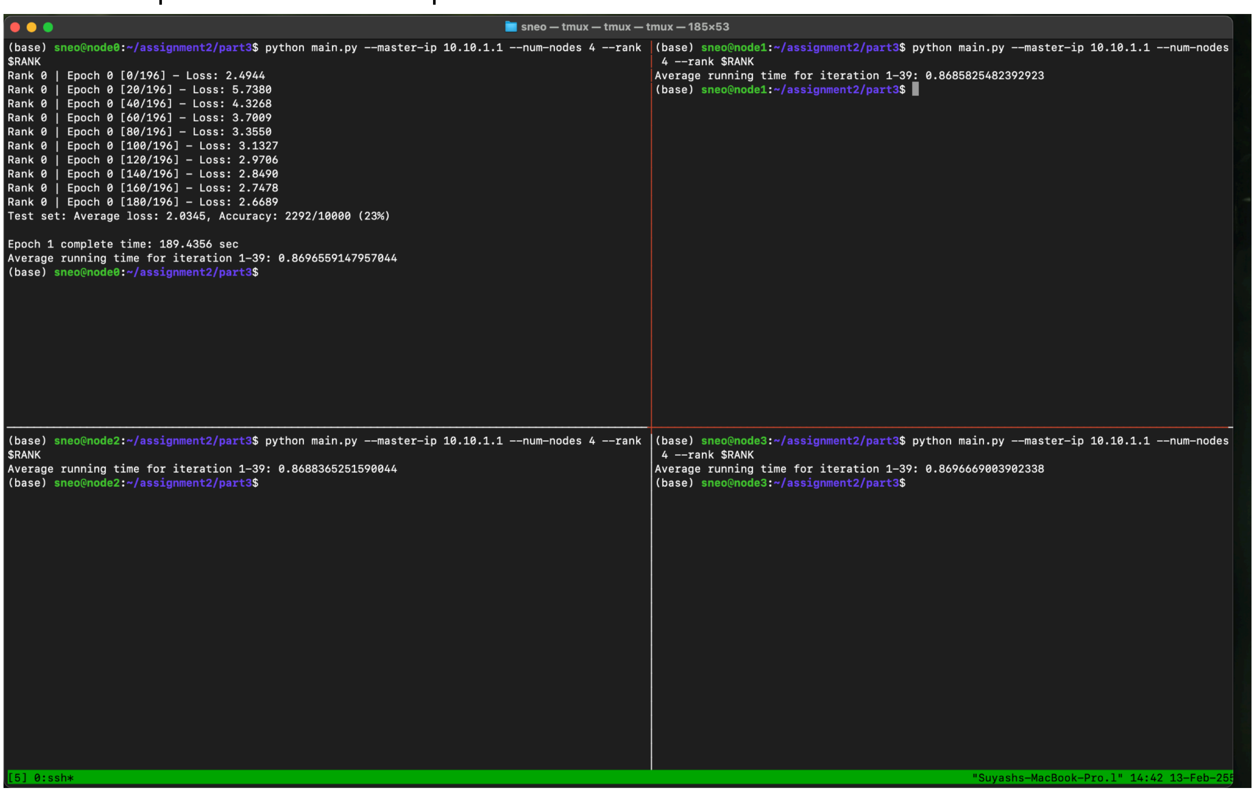




Part 3

In this part, we used the distributed functionality provided by PyTorch.

- Average loss: **2.0345**
- The test accuracy after 1 epoch was **23%**.
    - We didn't see a significant improvement in accuracy over just 1 epoch.
- Average time per iteration (for the first 40 iterations): **0.870 sec**
- Completion time for one epoch: **189 sec**



- Following is a network profile while running this job

- Difference between Figure 3.1 and 3.3 gives the following observation:
  - Data transmitted (node0) during job = 9.89 GB
  - Data received (node 0) during job = 9.87 GB
- The data is almost equal to but slightly less than observed in parts 2a and 2b
- Figure 3.2 shows top tx speed as 94540.69 KB/s
  - Not significantly different from our observations for previous parts

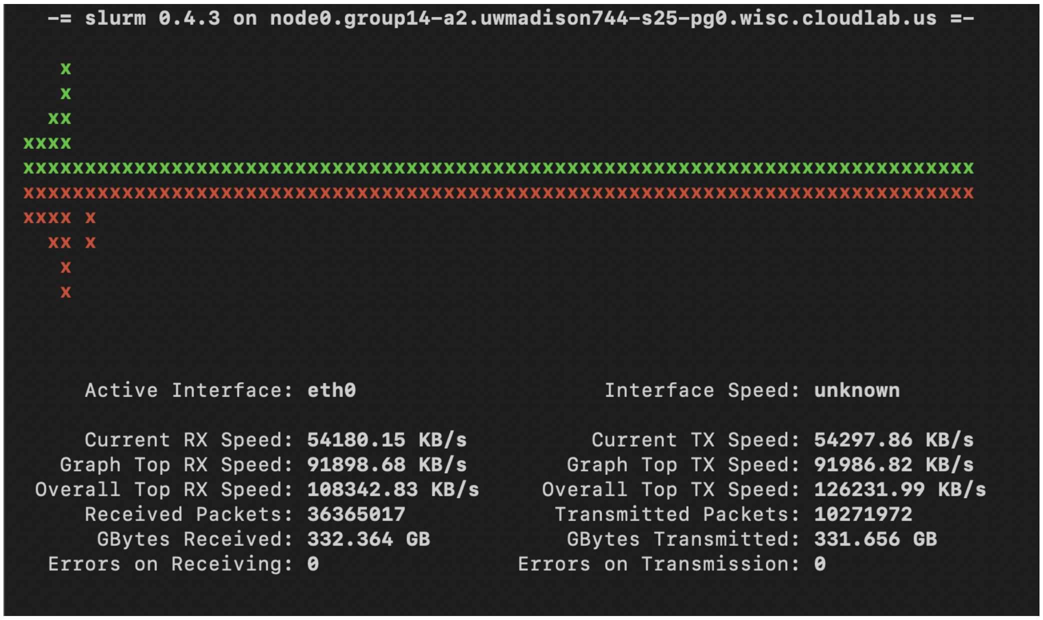


Figure 3.1 : Part 3 - Starting the job

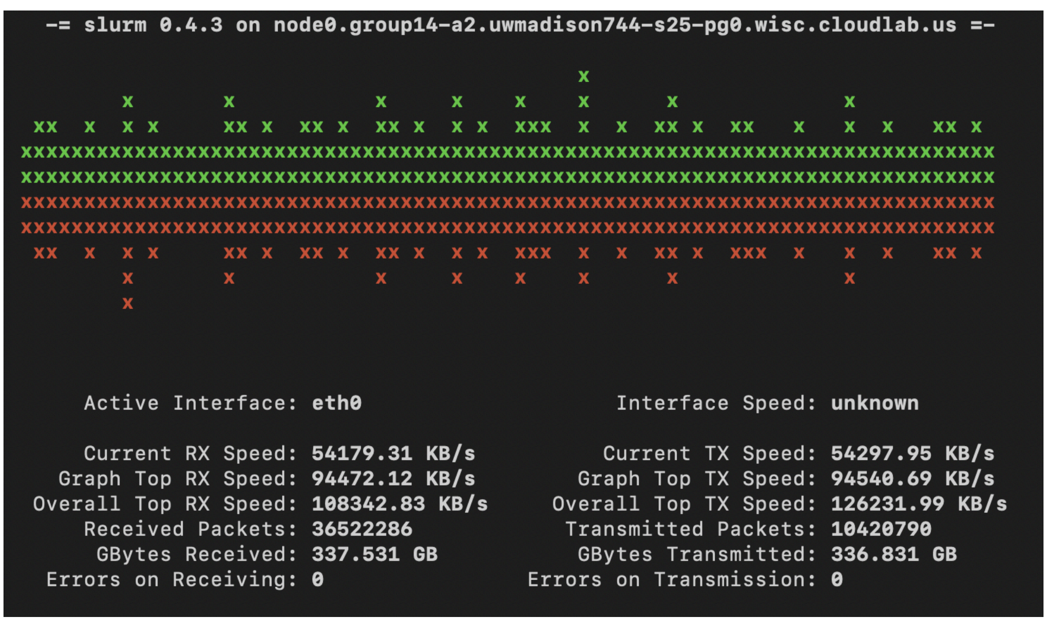


Figure 3.2 : Part 3 - During the job

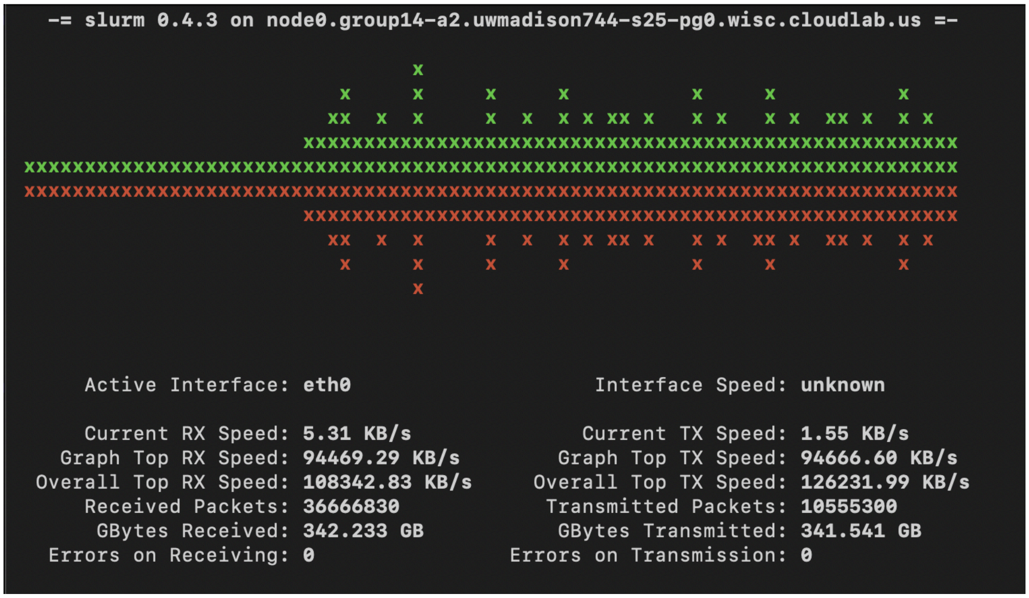
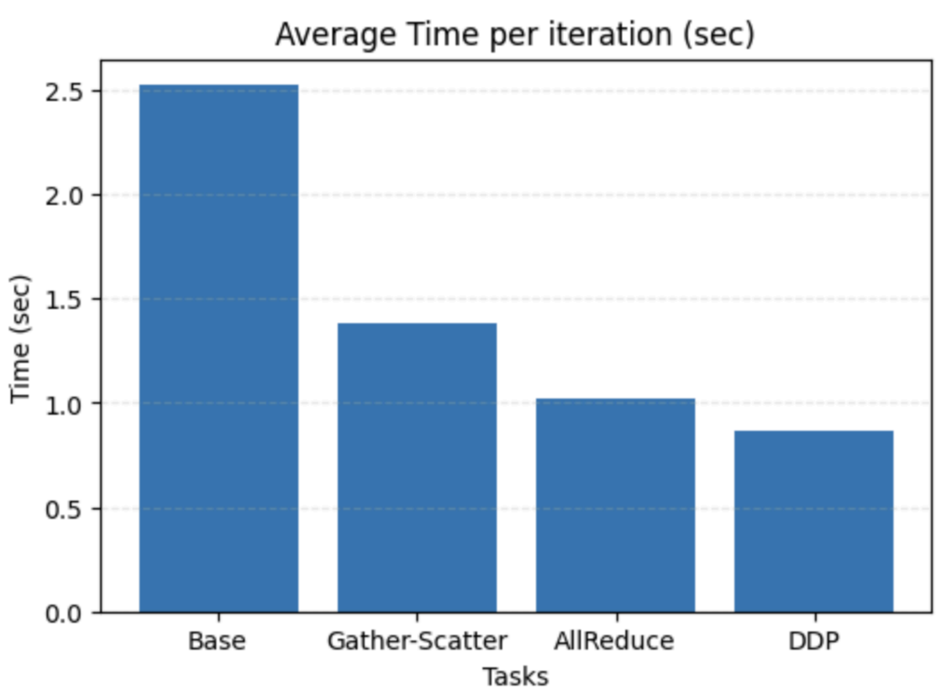


Figure 3.3 : Part 3 - After the job is finished

Conclusion

We ran each part for 40 iterations, discarded the timings of the first iteration and reported the average time per iteration for the remaining 39 iterations. By comparing the average time per iteration, we can evaluate the performance of different methods for training models.



As shown in the figure above, distributed data parallel training across multiple nodes using Gather-Scatter and AllReduce can significantly reduce training time, compared to the baseline approach on a single machine.

AllReduce method is more efficient than Gather-Scatter, because it synchronizes gradients among neighboring nodes, eliminating the need for a central node to collect and distribute updates.

Finally, PyTorch's in-built Distributed Data Parallel (DDP) achieves the fastest training time due to additional optimizations. Instead of sending each layer's gradient separately, DDP groups multiple gradients into buckets and sends them together. This reduces communication overhead and improves network efficiency, making it the most efficient method.

Contributions

All group members actively contributed to discussion, coding, and troubleshooting throughout the project.

- Suyash Raj: Part 1, Part 3
- Handan Hu: Part 1, Part 2b
- Amr Alazali: Part 1, Part 2a

For Part 1, Amr implemented the initial version, Suyash and Handan refined it by adding printing and timing functionalities. Additionally, each member was responsible for implementing one specific part of the assignment.