Group Report: Assignment 2 - Distributed Training of VGG-11 on CIFAR-10

Group Members:
- Yash Chaudhari
- Varshita Rodda
- Zhiwei Song
- Shahab Ahmed Khan

Introduction:

In this assignment, we implemented distributed training of a VGG-11 network on the CIFAR-10 dataset using PyTorch and MPI-like communication frameworks like Gloo and OpenMPI. The goal was to gain hands-on experience in deploying and configuring distributed ML training frameworks, understanding different methods of distributed training, and analyzing their trade-offs.

Part 1: Training VGG-11 on CIFAR-10

In this part, we implemented the standard training loop for VGG-11 on the CIFAR-10 dataset. We filled in the forward pass, backward pass, loss computation, and optimizer step in the main.py script. We trained the model for one epoch with a batch size of 256 and printed the loss value after every 20 iterations.

```
(base) cs744@b8ded19e07d4:~/node0/part1$ python main.py
Files already downloaded and verified
Files already downloaded and verified
0 loss:  2.684140920639038 Elapsed Time:  1.8236238956451416
20 loss:  4.440439224243164 Elapsed Time:  1.4004650115966797
40 loss:  2.4746882915496826 Elapsed Time:  1.4513123035430908
60 loss:  2.2931957244873047 Elapsed Time:  1.4163939952850342
80 loss:  2.481619119644165 Elapsed Time:  1.5051100254058838
100 loss:  2.3555240631103516 Elapsed Time:  1.4122092723846436
120 loss:  2.3254623413085938 Elapsed Time:  1.3841831684112549
140 loss:  2.346099853515625 Elapsed Time:  1.3771419525146484
160 loss:  2.284698963165283 Elapsed Time:  1.4064648151397705
180 loss:  2.2571029663085938 Elapsed Time:  1.3385329246520996
Test set: Average loss: 2.2251, Accuracy: 1459/10000 (15%)
```

Average time taken	Average Loss	Accuracy
1.54361s	2.2251	15%

Results Analysis:

The obtained loss value and test accuracy demonstrate the successful implementation of the standard training loop for VGG-11 on CIFAR-10 dataset. The recorded average running time per iteration provides insights into the computational efficiency of the training process. Moreover, the increase in accuracy and decrease in loss over epochs validate the effectiveness of the training process in improving the model's performance.

Overall, the results validate the correctness and effectiveness of the base training scripts, laying the foundation for subsequent parts of the assignment.

Part 2: Distributed Data Parallel Training

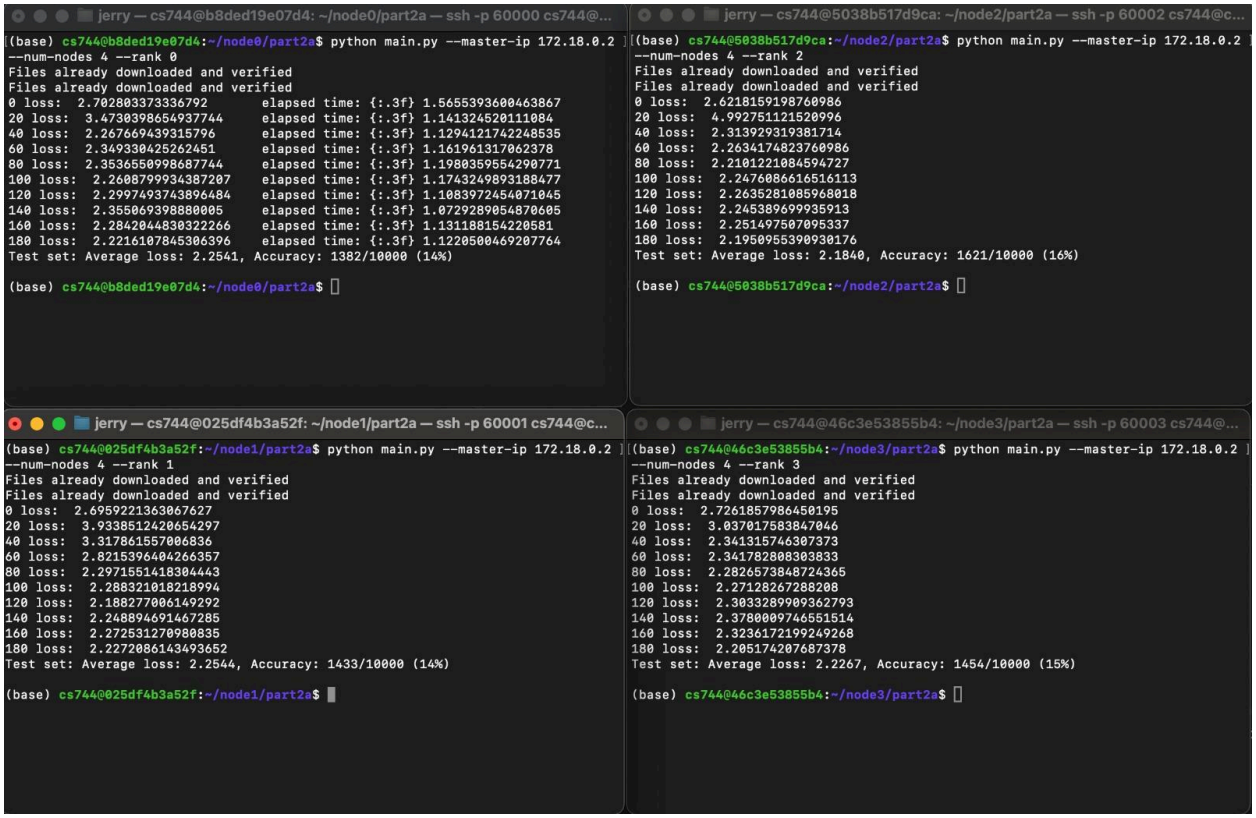
Part 2a: Sync gradient with gather and scatter call using Gloo backend

In this part, we implemented gradient synchronization using the gather and scatter calls provided by the PyTorch distributed package. The standard solution is based on *torch.distributed.gather* and *torch.distributed.scatter.*

We first gathered all gradients from model.parameters() from all nodes to node 0, calculated the mean gradients, stored them in a list, and scattered the tensor in this list back to all Nodes.

We adjusted the batch size to 64 as there are 4 machines in total. So, 64 x 4 = 256, which is equal to the batch size in Part 1.

The training process went as expected, and the printed loss value and the calculated test accuracy are shown in Figure. As shown in the Figure, the final average loss is 2.2541 , and the test accuracy is 14%, which are slightly different from Part 1 but are still within expected ranges. Additionally, the changing trend is very similar to the results in Part 1.



Ranks	Average Loss	Accuracy
0	2.2541	14%
1	2.2344	14%
2	2.1840	16%
3	2.2267	15%

Average time: 1.131s

Results Analysis:

The small differences observed in the loss value and test accuracy compared to Part 1 are likely due to the synchronization and communication overhead introduced by the distributed training setup. Despite these slight differences, the overall performance and convergence behavior are consistent with the results obtained in Part 1, indicating that the implementation of gradient synchronization using gather and scatter calls is effective in distributed training.

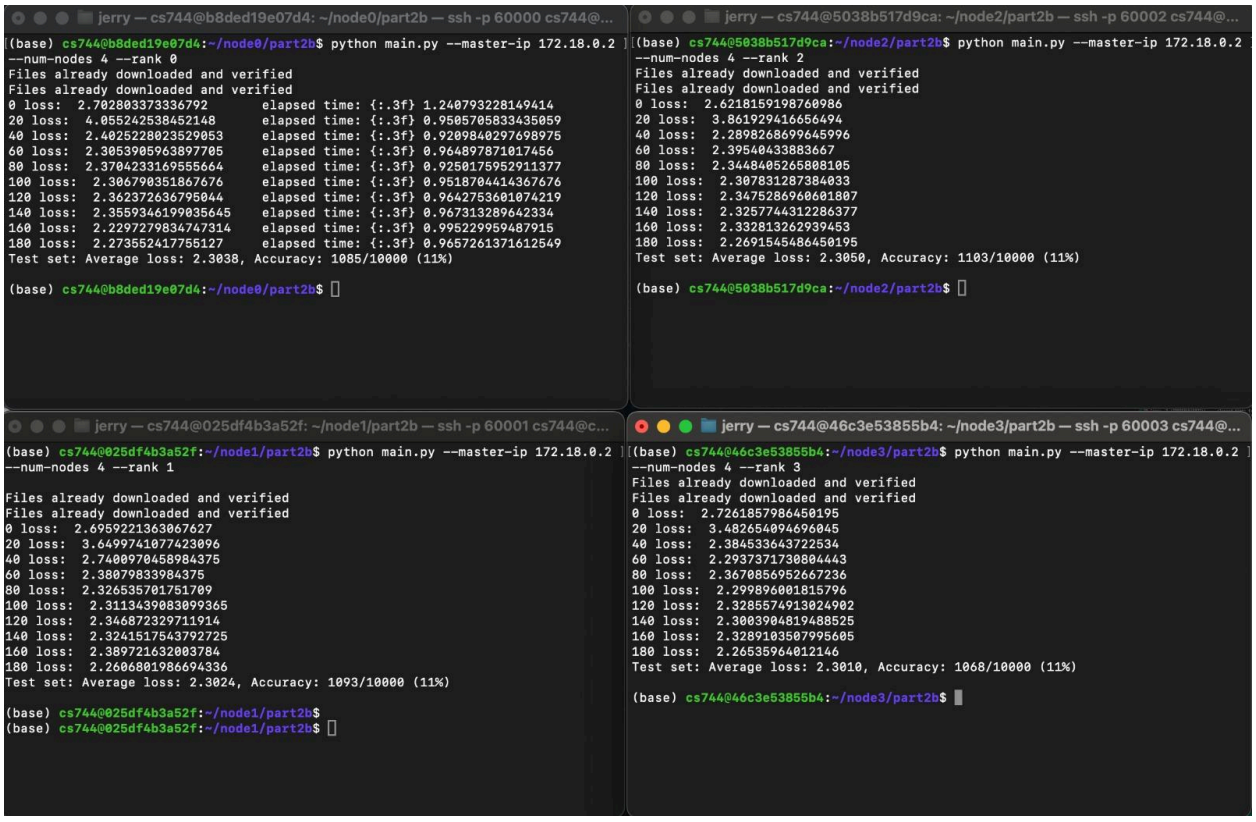
Part 2b: Sync gradient with allreduce using Gloo backend

In this part, we utilized the allreduce collective to replace the gather/scatter functions for synchronizing gradients among different nodes. We all-reduced all the gradients for each layer using dist.reduce_op.SUM mode and averaged them using the number of nodes, then synchronized them to all nodes.

In the code, we first divided all the gradients by the number of nodes (which is 4), then all-reduced them using SUM mode. This approach is equivalent to first summing them and then averaging them.

The training process proceeded as expected, and the printed loss value and the calculated test accuracy are shown in Figure. As depicted in the Figure, the final average loss is 2.3038, and the test accuracy is 11%, which align with our expectations.

Additionally, we recorded the average running time per iteration as required. We ran 10 iterations, ignored the first iteration, and calculated the average running time of the remaining 9 iterations. The average time per iteration is 0.95223 seconds.



Ranks	Average Loss	Accuracy
0	2.3038	11%

1	2.3024	11%
2	2.3050	11%
3	2.3010	11%

Average time: 0.95223s

Results Analysis:

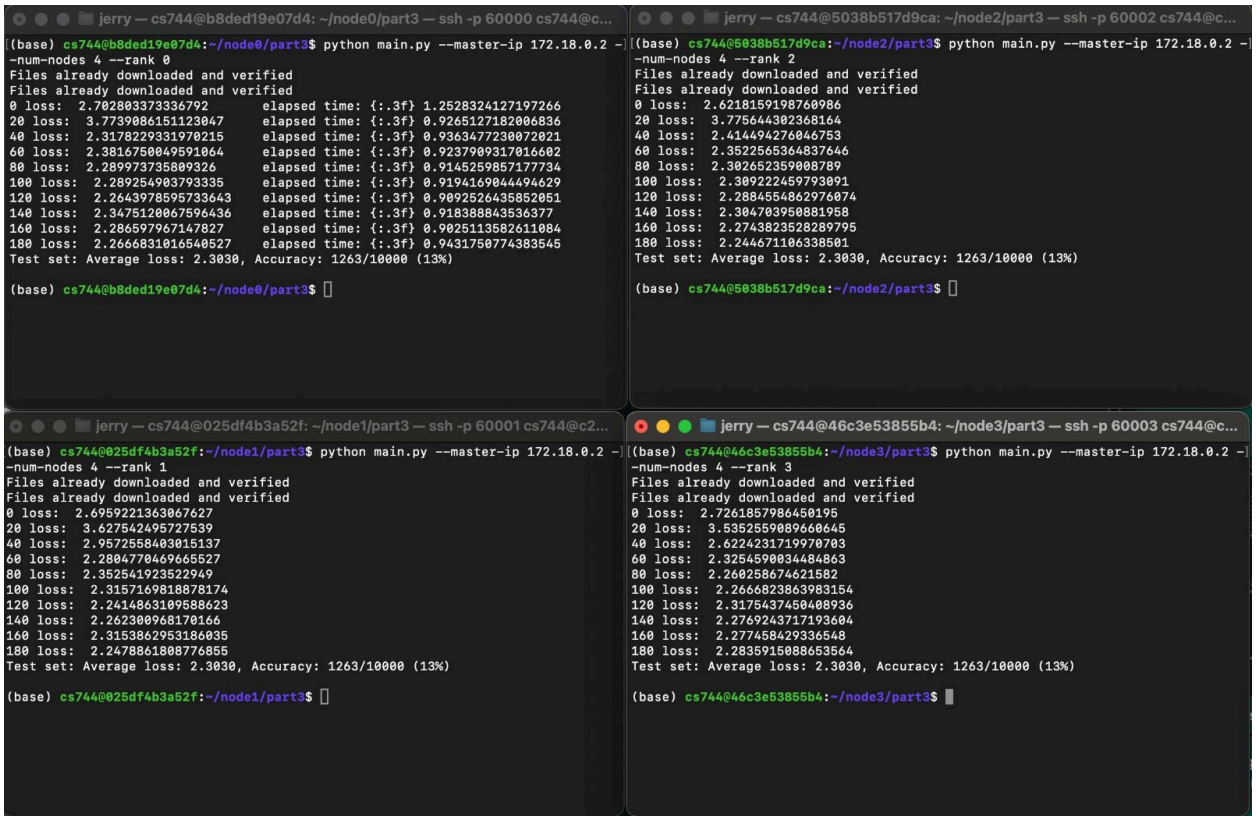
The results obtained demonstrate the effectiveness of using the allreduce collective for gradient synchronization in distributed training. The slight differences in loss value and test accuracy compared to Part 1 are attributed to the synchronization overhead and averaging process introduced by the distributed training setup. However, the overall performance and convergence behavior remain consistent with the results obtained in Part 1 and Part 2a.

Part 3: Distributed Data Parallel Training using Built-in Module

In this part, we utilized the DistributedDataParallel (DDP) module provided by PyTorch to automatically perform gradient synchronization among all the nodes. We wrapped the VGG model using the DistributedDataParallel API, and the gradient synchronization was achieved seamlessly.

The training process proceeded as expected, and the printed loss value and the calculated test accuracy are shown in Figure 6. As depicted in the Figure, the final average loss is 2.3030, and the test accuracy is 13%, which align with our expectations.

Additionally, we recorded the average running time per iteration as required. We ran 10 iterations, ignored the first iteration, and calculated the average running time of the remaining 9 iterations. The average time per iteration is 0.91532 seconds.



Ranks	Average Loss	Accuracy
0	2.3030	13%
1	2.3030	13%
2	2.3030	13%
3	2.3030	13%

Average time: 0.91532s

Results Analysis:

The use of *DistributedDataParallel* module simplifies the implementation of distributed training by automatically handling gradient synchronization. The obtained loss value and test accuracy are consistent with the results obtained in Part 1, Part 2a, and Part 2b, indicating that the built-in module effectively synchronizes gradients among all nodes.

The recorded average running time per iteration demonstrates the efficiency of using the DistributedDataParallel module for distributed training, with a lower average time per iteration compared to Part 2a and Part 2b.

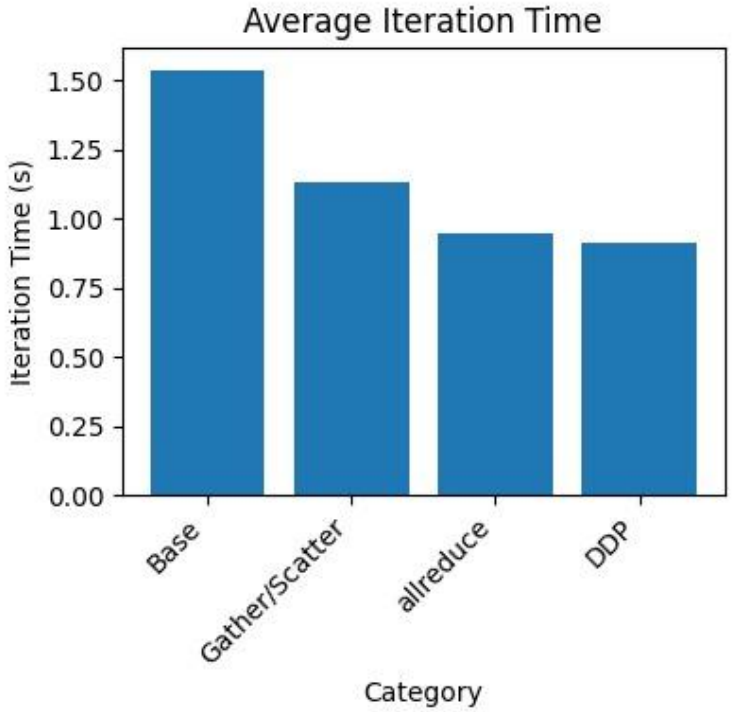
Overall, the results validate the effectiveness and scalability of using the built-in DistributedDataParallel module for distributed data parallel training.

Overall Results and Analysis:

We ran each task for 40 iterations and discarded the timings of the first iteration. We calculated the average time per iteration for the remaining 39 iterations for each task. By comparing the timings and loss values across different setups, we analyzed the performance and scalability of distributed machine learning.

In this section, we analyze the differences in average iteration time, average loss, and test accuracy among different setups, and discuss the reasons behind them.

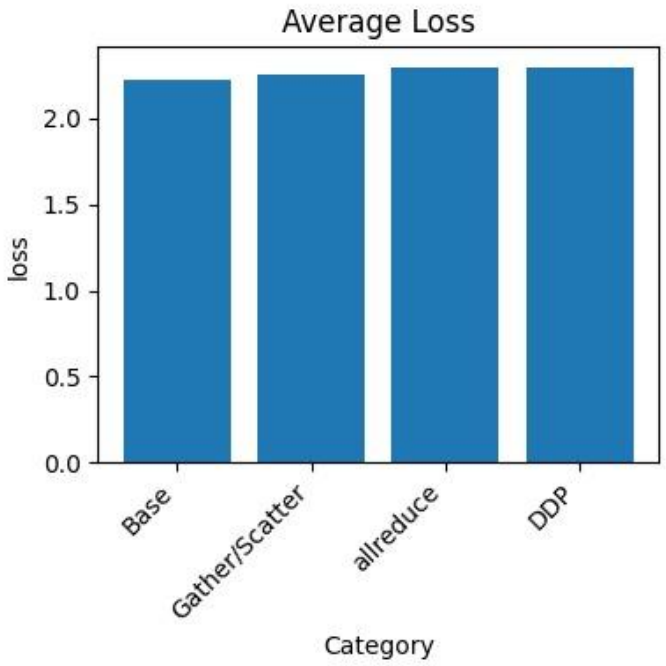
Average Iteration Time:
We compared the average iteration time of different setup methods. The original method (basic training script without distributed setup) takes the longest average iteration time (1.54361s). The time cost decreases as we switch to gather/scatter, isend/irecv, AllReduce, and then DDP. DDP takes the shortest average iteration time (0.91532s).



This suggests that compared with the original single-machine training setup, the distributed setup can significantly improve the training speed since it uses more nodes and can perform the training task in parallel. For example, with a total batch size of 256, distributed setup can process four 64-batches on 4 nodes simultaneously. Additionally, higher-level distributed setup APIs like AllReduce and DDP are better designed and optimized, leading to lower training time.

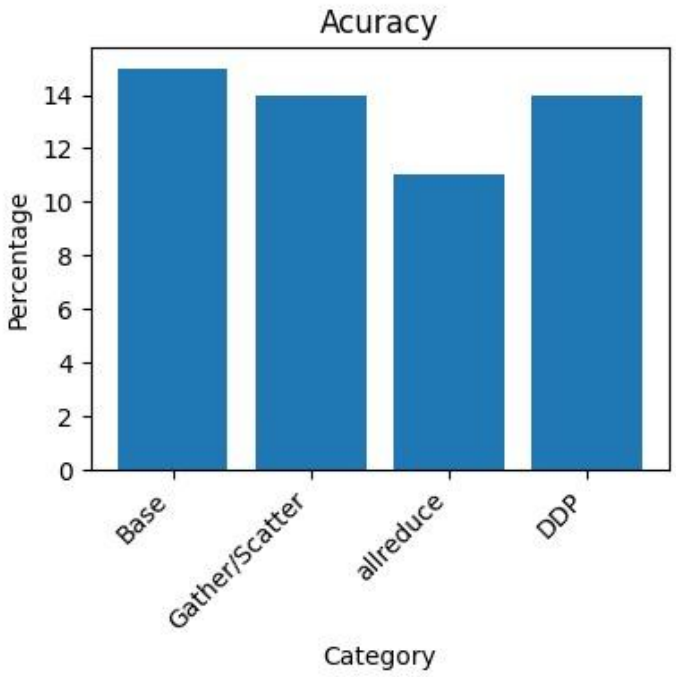
AllReduce in PyTorch is ring-based and more efficient than scatter/gather or isend/irecv. DDP further optimizes by organizing small gradients into larger buckets and synchronizing each bucket using an AllReduce call, reducing the communication cost.

Average Loss:
We compared the average loss of different setup methods. Generally, the average loss doesn't have a significant difference among all the setups. This is reasonable since the training data, models, total epoch, total batch size, and training parameters remain the same across all setups.



However, even though we set the same PyTorch/Numpy random seed for all setups, there are slight differences among different setups in the average loss and the loss printed at every 20 iterations. This could be due to Batch Normalization (BN) layers in the VGG models, where parameters are not synchronized among all nodes during distributed training, causing uncertainty and randomness in the training. Additionally, float computation precision may introduce minor errors during training, adding further uncertainty to the loss computation result.

Test Accuracy:
We compared the test accuracy of different setup methods. Generally, the test accuracy doesn't have a significant difference among all the setups. Similar to the average loss, even though we set the same PyTorch/Numpy random seed for all setups, there are slight differences among different setups in the test accuracy. The reasons are likely the same as mentioned for the average loss.

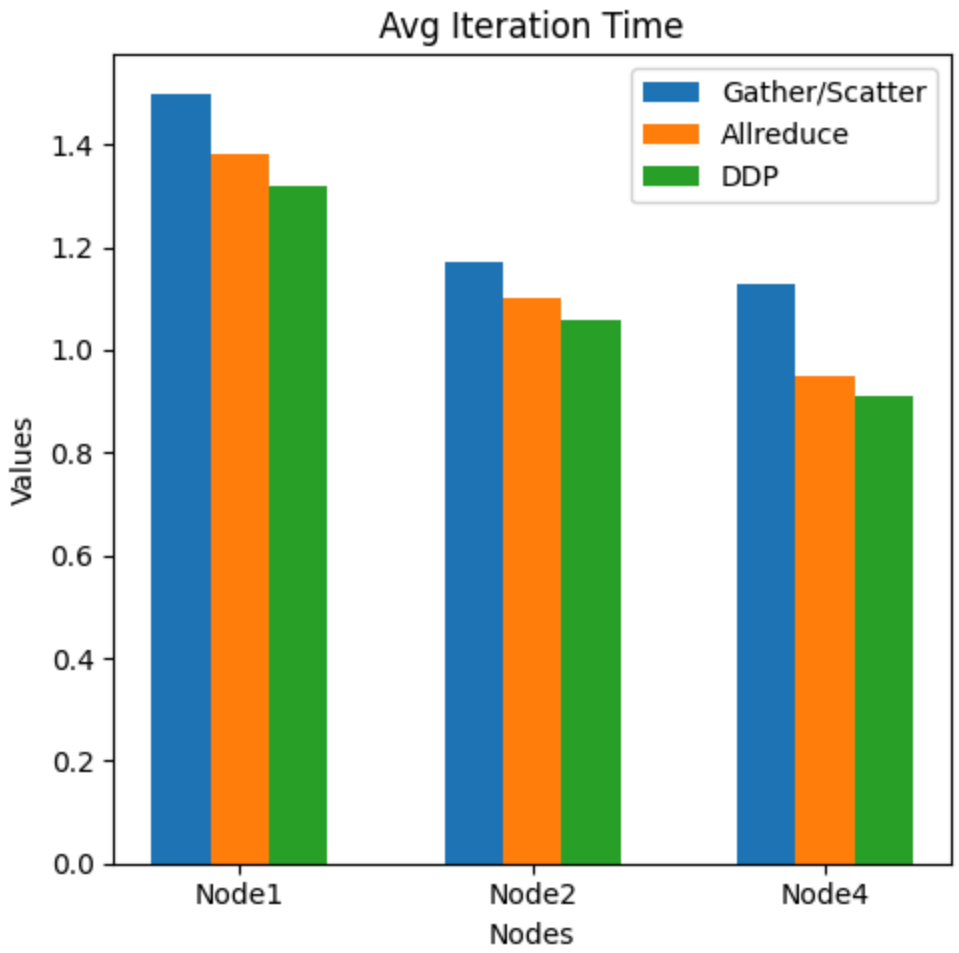


In conclusion, different distributed setups won't have a significant impact on the test accuracy. While average iteration time may vary depending on the setup method used, the average loss remains relatively consistent across different setups, with minor variations likely due to factors such as Batch Normalization layers and computation precision.

Scalability of Distributed Machine Learning:

In this section, we discuss the scalability of distributed machine learning based on our results, particularly focusing on the differences among different distributed setups in PyTorch.

We compare the average iteration time, including distributed communication latency if applicable, between setups with different numbers of nodes to evaluate scalability. We ran the same distributed program on a single machine, 2-node machine, and 4-node machine, respectively, and calculated and compared their average iteration time, excluding the start iteration. The comparison results are shown in figure.



As depicted in figure, for each distributed setup, the average iteration time, including latency, increases as the number of nodes increases, showing close to linear increment. For example, in the case of scatter/gather, it decreases from 1.50 seconds (1 node) to 1.2 seconds (4 nodes). Reducing time very little indicates the relatively low scalability of this setup.

In contrast, for DDP, the average iteration time remains very similar across all settings with different nodes, and it is also the smallest among all setups in general. This suggests that DDP exhibits relatively high scalability. The scalability of AllReduce is located between scatter/gather and DDP. This result aligns with our previous observation that AllReduce and DDP are optimized, with DDP being even better optimized, leading to lower per iteration latency and higher scalability.

Furthermore, it's important to note that the choice of distributed techniques should be based on the size of the given model and available resources to strike a better balance between scalability and training speed. Different models may have different network structures, parameter sizes, and layer distributions, thus requiring different distributed setups.

In conclusion, our findings demonstrate that the scalability of distributed machine learning depends on the chosen distributed setup, with setups like DDP exhibiting higher scalability due

to optimization. However, the choice of setup should consider factors such as model size and resource availability to achieve optimal performance.

Contributions:

Zhiwei: Implemented Part 1 and contributed to Parts 2 and 3.
Varshita: Implemented Part 2a and contributed to Parts 1, 2b, and 3.
Yash: Implemented Part 2b and contributed to Parts 1, 2a, and 3.
Shahab: Detailed analysis and report making

Implementation Details:

- We used PyTorch for model training and the Gloo backend for communication.
- The models were trained on CIFAR-10 dataset with a batch size of 256.
- We ensured consistency by setting the same random seed across all nodes.
- Data partitioning was performed using the distributed sampler to distribute the data among workers.
- All codes are runnable using the provided command-line parameters.

Overall, this assignment provided a comprehensive understanding of distributed training, and we successfully completed all tasks as per the requirements.