FALL 2021 CS744 Assignment 2

Aditya Kaushik Kota Isha Padmanaban Roopa Malavika Daripally

1 Objective

The aim of the assignment is to get experience working with PyTorch, MPI, Gloo and OpenMPI and understand the benefits and shortcomings of different approaches of performing distributed training and their scalability.

2 Introduction

PyTorch was initially developed by Facebook's AI Research lab. This machine learning library is available as open source implementation. It is based on the Torch library. Main applications for PyTorch are found in the fields of machine learning, big data computer vision and natural language processing.

As the name suggests, in distributed training, the workload to train a model is divided and distributed across multiple worker nodes. This is done to speed up the training process. The main applications of distributed training is found for compute heavy tasks like training deep neural networks.

Message Passing Interface (MPI) is one of the communication protocols used in the instances of parallel programming. This allows the parallelized implementation of applications across number of separate computers connected by a network.

Gloo library supports a collection of communications. It provides support for a number of collective algorithms such as allreduce, barrier, and broadcast. These algorithms are useful for machine learning applications.

3 Experimental setup and Installation

A Cloudlab experiment is created with a four node cluster. Ubuntu is installed in each machine. Created conda environment to install PyTorch and related packages.

4 Experiments

4.1 Part 1

Experiment: Training on VGG-11 with Cifar10
We trained on VGG-11 network with Cifar10 dataset which has 60000 color images with 32x32 dimensions split in 10 classes thus consisting of 6000 images per class. Training images are 50000 in number and test images are 10000 in number. We added the standard training parameters with forward pass, backward pass, loss computation and optimizer step.

Observations: With a batch size of 256 samples, which runs for 196 iterations approximately for 1 epoch, we observed the following results :

```
20 loss: 2.636019468307495
40 loss: 2.3483173847198486
60 loss: 2.3193373680114746
80 loss: 2.2596027851104736
100 loss: 2.2726035118103027
120 loss: 2.3302791118621826
140 loss: 2.23667311668396
160 loss: 2.1521780490875244
180 loss: 2.076960563659668
Test set: Average loss: 2.1339, Accuracy: 1935/10000 (19%)
Average Time per iteration for 39 iterations : 1.611695872 secs
```

4.2 Part 2a Gather and Scatter

Experiment: Sync gradient with gather and scatter call using Gloo backend
We perform the aggregation using gather and scatter communication collectives. Specifically Rank 0 in the group will gather the gradients from all the participating workers and perform elementwise mean and then scatter the mean vector. The workers update the gradient variable with the received vector and then continue training.

Observations: With a batch size of 64 samples across 1 master and 3 worker nodes, which runs for 196 iterations approximately for 1 epoch, we observed the following results :

```
20 loss: 3.5594327449798584
40 loss: 2.2740800380706787
60 loss: 2.3407247066497803
```

```
80 loss: 2.2692365646362305
100 loss: 2.2818896770477295
120 loss: 2.2219715118408203
140 loss: 2.1900374889373 78
160 loss: 2.1110215187072754
180 loss: 2.1038448810577393
Test set: Average loss: 2.1436, Accuracy: 1783/10000 (18%)
Average Time per iteration for 39 iterations : 1.25639041 secs
```

4.3 Part 2b All Reduce

Experiment: Sync gradient with allreduce using Gloo backend
Ring Reduce is a scalable technique for performing gradient synchronization. Instead of working with scatter and gather collectives separately, the built in allreduce collective can be used to sync gradients among different nodes. The gradients are read after backward pass layer by layer and allreduce is prerformed on the gradient of each layer. PyTorch allreduce call doesn't have an 'average' mode, the 'sum' operation is used to then get the average on each node by dividing with number of workers participating. After averaging, the gradients of the model are updated.

Observations: With a batch size of 64 samples across 1 master and 3 worker nodes, which runs for 196 iterations approximately for 1 epoch, we observed the following results :

```
20 loss: 2.655669689178467
40 loss: 2.5256664752960205
60 loss: 2.27115797996521
80 loss: 2.232382297515869
100 loss: 2.261044979095459
120 loss: 2.1849939823150635
140 loss: 2.2463133335113525
160 loss: 2.06764817237854
180 loss: 2.0221049785614014
Test set: Average loss: 2.1378, Accuracy: 1874/10000 (18.74%)
Average Time per iteration for 39 iterations - 0.9883078718 secs
```

4.4 Part 3 Distributed Data Parallel

Experiment: Distributed Data Parallel Training using Built in Module
Distributed Data Parallel (DDP) provides multi-process parallelism where the processes can be across

different machines. The process group should be initialised and multiple processes are spawned. When DDP is invoked for a model, an instance of DDP is created per process.

The parallelization of the model execution is achieved by dividing the input across the specified workers with the help of batch size parameter. And replication of the model on each worker is accomplished where each of the replica will now run a part of the input.

The gradients from each worker is averaged after completing the backward pass. And the gradients are communicated to synchronise the model replicas. The Global Interpreter Lock (GIL) contention across model replicas is not present in DDP.The training is speeded up because the model is broadcasted during construction of DDP itself rather than in every forward pass. Also, gradient computations are overlapped with gradient communications in DDP.

The forward pass, backward pass, and optimizer step are executed on the DDP model and the results are summarised below. DDP is generally faster and provides better accuracy.

Observations: With a batch size of 64 samples across four workers which runs for 196 iterations approximately for 1 epoch, we observed the following results:

```
20 loss: 2.492549180984497
40 loss: 2.53298282623291
60 loss: 2.277205228805542
80 loss: 2.2005510330200195
100 loss: 2.224674701690674
120 loss: 2.178819417953491
140 loss: 2.169401168823242
160 loss: 1.995828628540039
180 loss: 1.9988471269607544
Test set: Average loss: 2.0405, Accuracy: 2087/10000 (20.87%)
Average Time per iteration for 39 iterations : 0.7678834103 secs
```

5 Evaluations

5.1 Comparison of CPU/Bandwidth/Memory Statistics across Task-2a, Task-2b and Task-3 for Master Nodes

From Figure: 1, we can infer that CPU utilization for Gather-Scatter is less compared to All Reduce and PyTorch's DDP as it is waiting for the master to compute gradient and is not leveraging the available CPU at that time. DDP overlaps computation of gradients with communication, therefore has highest CPU utlization and completes its execution faster when compared to others.

From Figure: 2, its evident that since we are using same training dataset, there is no significant difference found in memory utilization across different Gradient Synchronization Techniques.

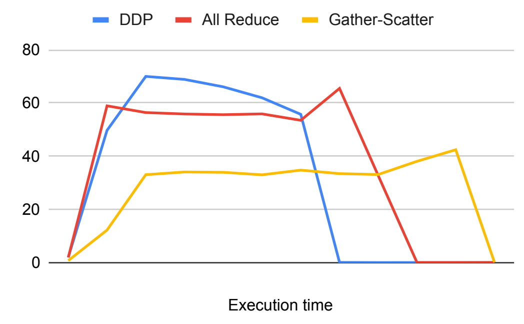

Figure 1: Percentage of CPU Utilization for different Gradient Synchronization Techniques

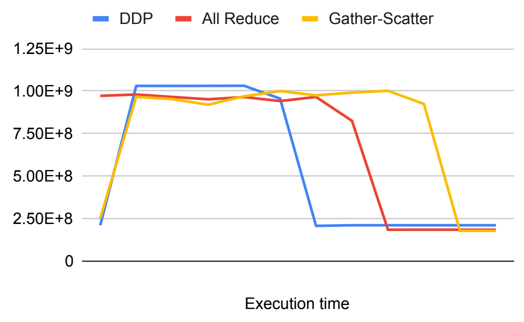

Figure 2: Memory Utilization for different Gradient Synchronization Techniques

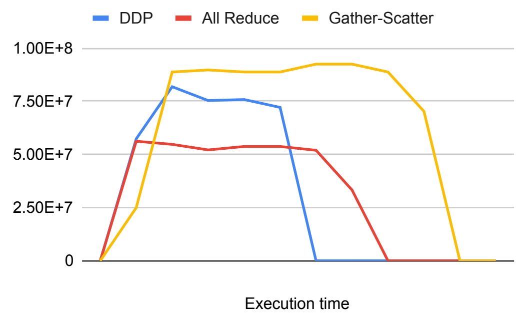

Figure 3: Network Statistics for different Gradient Synchronization Techniques

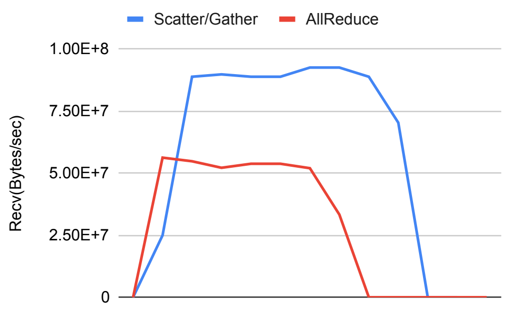

Figure 4: Bandwidth Recv Bytes across Scatter/Gather and AllReduce

From Figure: 3, we can observe Gather-Scatter has more network statistics in terms of number of bytes sent and received when compared to DDP and All Reduce. This is because each worker has to scatter and gather gradients in case of Gather-Scatter whereas in All Reduce and DDP each worker scatter and computes the average on its own therefore less calls. DDP does have more calls compared to All Reduce since it overlaps communication with computation reducing overall execution time but with considerable network calls for synchronizing gradients.

5.2 Comparison of Network Bandwidth Statistics across Scatter/Gather and AllReduce for Master nodes

We are comparing the number of bytes received on the network interfaces across Scatter/Gather and AllReduce gradient reduction methods and we can observe that Scatter/Gather is taking more bandwidth comparatively as shown in Figure 4. This can be explained by the reasoning that in Scatter/Gather, the master is receiving gradients from all the worker nodes, performing the average and then sending back the gradients across network to all the worker nodes whereas in AllReduce the averaging of the gradients is being done across all nodes at the same time and requires less communication overhead.

5.3 Comparison of Average Iteration Time and CPU/Memory/Bandwidth statistics for Task-3

Each DDP process creates a reducer that is local to itself to synchronize gradients during the backward pass. This reducer can also improve the communication efficiency by efficiently organising the gradients into buckets, and reducing one bucket at any point of time. Bucket size setting can be changed with the parameter bucket_cap_mb.

As bucket size capacity is increased as 5mb, 10mb, 25mb, and 50mb, we can observe in Figure 7, that the total amount of bytes received (recv stats) on network interfaces is reduced. The same trend is observed for the total amount of bytes sent (send stats) on network interfaces as bucket sizes are increased. This shows how the communication overhead reduces as the bucket sizes are

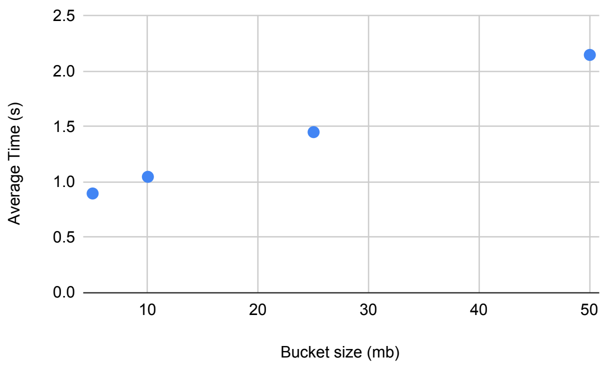

Figure 5: Average Iteration time when increasing Bucket Size for DDP

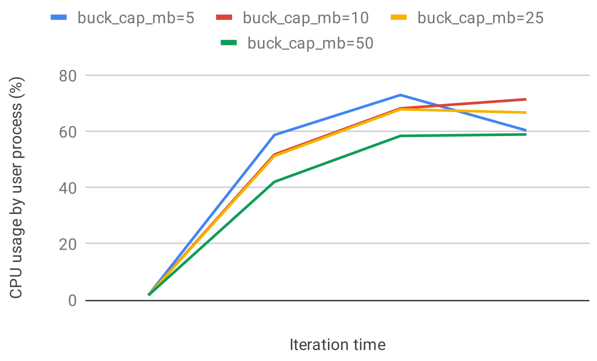

Figure 6: Percentage of CPU used by user process when increasing Bucket Size for DDP

increased. Hence the average iteration time is also increased as shown in Figure 5 as the bucket sizes are increased.

5.4 Comparison of Average Iteration Time across Task-1, Task-2a, Task-2b and Task-3

We ran the experiments using fixed total batch size of 256 per iteration for different tasks. From Figure 8, the Part 1 has highest iteration time with no distributed workload. Part-2a takes longer time to Part-2b since there is computation and network overhead on master to scatter the computed gradients. Part-3 takes less average time per iteration as the training is speeded up due to overlapping of gradient computation with gradient communications.

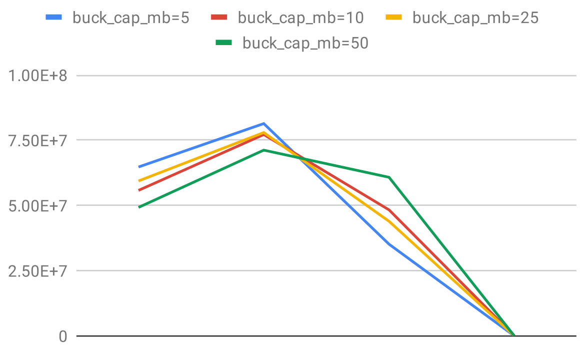

Figure 7: Bytes received on network interfaces when increasing Bucket Size for DDP

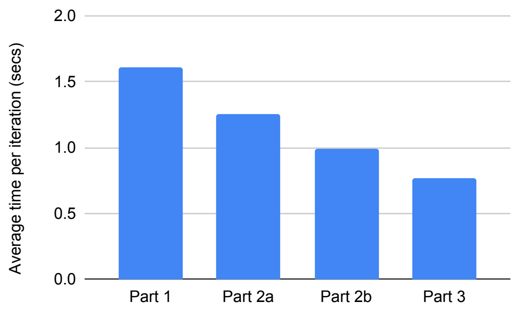

Figure 8: Average Iteration time for Part-1, Part-2a, Part-2b, Part-3

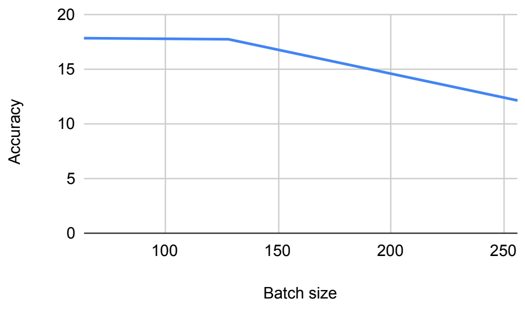

Figure 9: Accuracy for Gather Scatter across various batch sizes

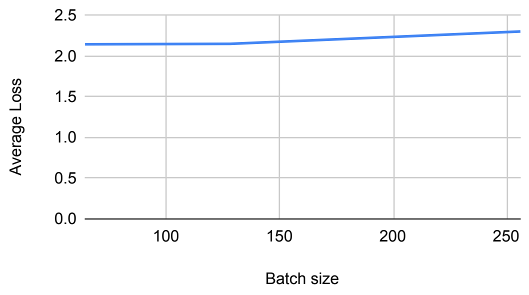

Figure 10: Loss for Gather Scatter across various batch sizes

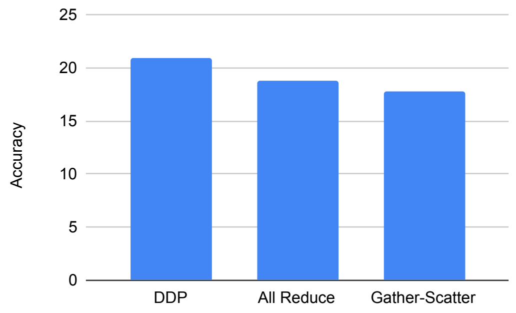

Figure 11: Model Loss with different Gradient Synchronization Techniques

5.5 Comparison of Loss and Accuracy across different batch sizes for Gather-Scatter

We are comparing accuracy and loss across various batch sizes - 64,128 and 256 for Scatter/Gather in Figures 9 and 10. We can see that as the batch size is increased, accuracy decreases. This is because the number of iterations decrease and thus the back propagation is reduced. This also leads to increase in the average loss.

5.6 Comparison of Loss and Accuracy across different Gradient Synchronization Techniques

We are comparing the accuracy across Scatter/Gather, All Reduce and DDP for a fixed batch size of 64 across the 4 nodes (master and slaves) as shown in Figure 12. We can see the accuracy is highest for DDP, followed by All Reduce and Scatter/Gather. This can be reasoned about by the fact that gradient synchronization and update is being done in DDP with optimized overlap of computation and communication whereas in others since being done separately, they may incur some loss of information during the communication across the various worker nodes. For the similar reasoning, we can observe an increased loss from DDP to All Reduce and Scatter/Gather as shown in Figure 11

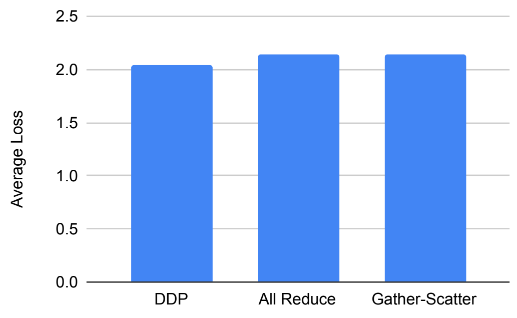

Figure 12: Model Accuracy with different Gradient Synchronization Techniques

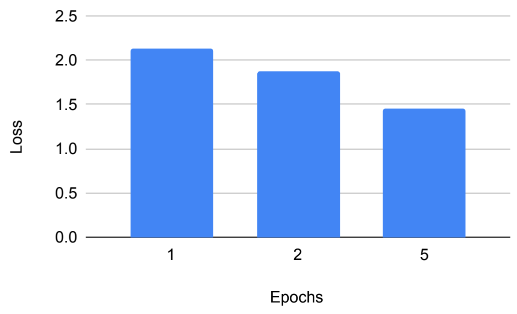

Figure 13: Model Accuracy with different number of epochs

5.7 Comparison of Loss and Accuracy with varying number of Epochs

We performed distributed training of the model for multiple epochs and computed loss and accuracy. As shown in Figures 13 and 14, the loss decreases as expected across the epochs. Thus the accuracy is also increased with increasing number of epochs.

5.8 Synchronous Vs Asynchronous update of gradients

We ran experiments by setting and unsetting the flag for async_op while updating the gradients during the backward pass which corresponds to Synchronous and Asynchronous updates respectively. With the former, all model replicas collectively communicate and reduce the gradients or parameters, while the asynchronous scheme employs P2P communication to update gradients or parameters independently.

Comparing the results, going from Synchronous to Asynchronous, we can observe that the accuracy decreases, loss increases and the average iteration time decreases as shown in Figure 15. This can be reasoned about saying that since asynchronous does more work in less time and also the gradients are not being updated synchronously, the accuracy decreases and loss increases whereas time decreases. Also comparing the bandwidth received bytes of data across the network during the course of the algorithm, we can see that asynchronous sends more data across, since there are more operations

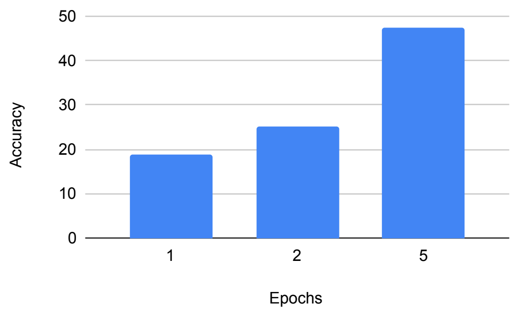

Figure 14: Model Loss with different number of epochs

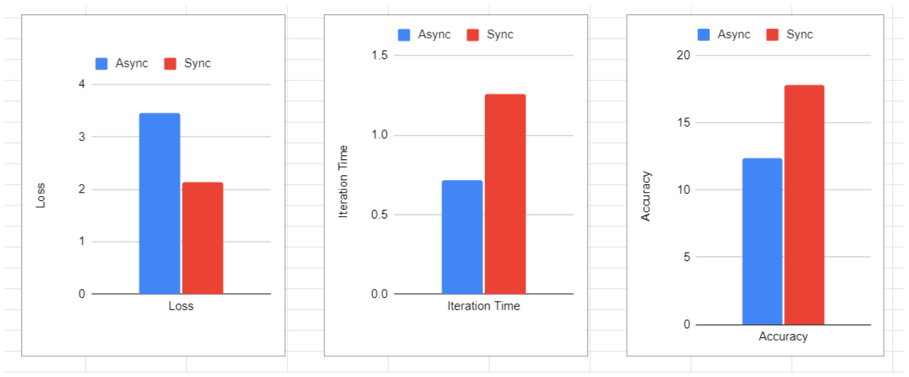

Figure 15: Loss/Accuracy/Iteration Time Async Vs Sync

being done in less amount of time as shown in Figure 16.

6 Contributions

1. Setting up environment for PyTorch - Roopa
2. Part 1 - Roopa
3. Part 2a - Roopa
4. Part 2b - Kaushik
5. Part 3 - Isha
6. Evaluation - Kaushik, Isha, Roopa
7. Report - Kaushik, Isha, Roopa

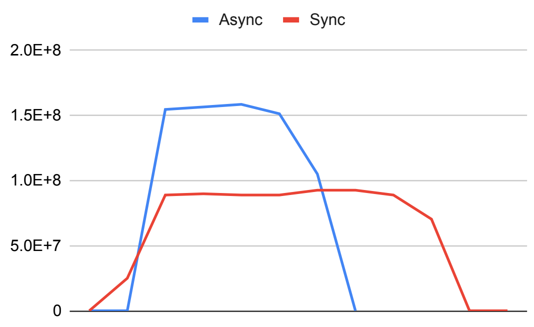

Figure 16: Bandwidth Recv Bytes Async Vs Sync

7 Conclusion

We got hands-on experience working with PyTorch. We learned different collective communication frameworks like Gloo, MPI and analyzed different Gradient Synchronization Techniques for performing distributed parallel training.

References

1. PyTorch Distributed: Experiences on Accelerating Data Parallel Training, https://arxiv.org/pdf/2006.15704.pdf

2. Technologies behind Distributed Deep Learning: AllReduce, https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce

3. Distributed Training in PyTorch — Part 1 (Distributed Data Parallel), https://medium.com/analytics-vidhya/distributed-training-in-pytorch-part-1-distributed-data-parallel-ae5c645e74cb

4. PyTorch Distributed Data Parallel, https://pytorch.org/docs/master/generated/torch.nn.parallel.DistributedDataParallel.html