CS744 - Big Data Systems
Assignment 2

Group 23:
Akul Gupta
Chahak Tharani
Deepti Rajagopal

Index

Implementation details

Part 1: Training VGG-11 on Cifar10

We wrote the functionality to train the model on a single CPU node for Task 1. In the training function, we loop over our data iterator, feed the inputs to the network and try to converge to optimal parameters.

1. *model.train()* tells the model that we are training the model. So effectively layers like dropout, batchnorm etc. which behave differently on the train and test procedures know what is going on and hence can behave accordingly.
2. We set our hyperparameters first, we then train and optimize our model with an optimization loop. Loss function measures the degree of dissimilarity of the obtained result to the target value, and we try to minimize this loss function during training.
3. We pass our model's output logits to *nn.CrossEntropyLoss*, which will normalize the logits and compute the prediction error.
4. We call *optimizer.zero_grad()* to reset the gradients of model parameters. Gradients by default add up; hence to prevent double-counting, we explicitly zero them at each iteration.
5. We backpropagate the prediction loss with a call to *loss.backward()*. PyTorch deposits the gradients of the loss w.r.t. each parameter.
6. Once we have the gradients, we call *optimizer.step()* to adjust the parameters by the gradients collected in the backward pass.

Part 2a: Distributed Data Parallel Training using Gather & Scatter Collectives

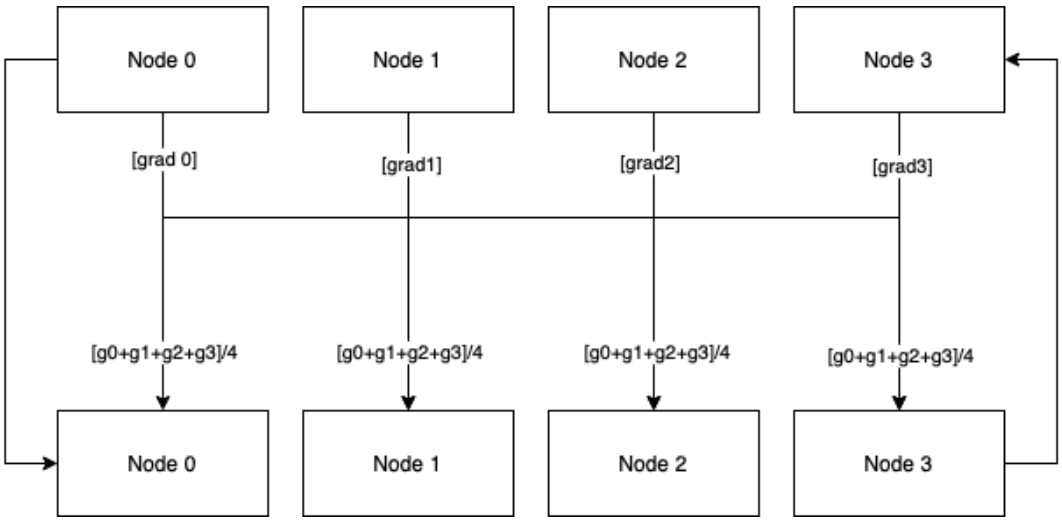
We made use of the script written in Part 1, and modified the code to implement distributed data parallel training by using the scatter and gather collectives to synchronize gradients. Distributed Data Parallel is set up over TCP, by passing the master's IP, number of nodes, and the rank of the specific node. Because we're following data-parallel execution, we need to ensure that the model is the same across all the nodes. This is done by ensuring that at the end of each iteration, every node has the same gradients. This is where the gather & scatter collectives come into play. The gather collective gathers all the gradients from all the processes into the master process. Scatter on the other hand is used to distribute a tensor from the master process on to all the other processes.

The following are the steps we followed to implement Part 2a:

1. Setup Pytorch in distributed mode using pytorch's *torch.distributed.init_process_group()*. This blocks until all processes have joined.
2. Partitioned our data using torch's *torch.utils.data.distributed.DistributedSampler()* based on world size of 4, and with *torch.maunal_seed* as 70.
3. For each of the parameters in *model.parameters()*, we tracked the gradients, used *torch.distributed.gather()* collective to gather all gradients to node0, and then scattered the mean of this gradient vector to all the nodes using *torch.distributed.scatter()*.
4. Since node0 is where we're gathering all the gradients and scattering to other nodes, we have an if-else block that branches out differently for rank 0 and the other ranks. This is because, only the node at which gather takes place, and the node from which scatter takes place, is where you need to provide non-"None" gather_list and scatter_list arguments.
5. We use torch.manual_seed() to seed the Random Number Generator for all devices
6. We disabled the benchmarking feature with torch.backends.cudnn.benchmark = False which causes cuDNN to deterministically select an algorithm, possibly at the cost of reduced performance.
7. The rest of the steps are similar to part 1.

Part 2b: Distributed Data Parallel Training using allreduce

In part 2b, we made use of all_reduce() instead of manual calls to gather() and scatter() in order to circulate the averaged gradients across nodes. All the gradients were summed and then averaged (sum / divided by the number of workers).



The following are the steps we followed to implement Part 2b:

1. Setup Pytorch in distributed mode using *torch.distributed.init_process_group()*.
2. The data was partitioned using DistributedSampler.
3. For each parameter, the gradients were summed using all_reduce() and then divided by the number of workers to average them.
4. The rest of the steps are similar to part 1.

Part 3: Distributed Data Parallel Training using Built in Module

We implemented distributed data parallelism based on the *torch.distributed* package at the module level. This container parallelises the application of the given module by splitting the input across the specified devices by chunking in the batch dimension. The module is replicated on each machine and each device, and each such replica handles a portion of the input. During the backwards pass, gradients from each node are averaged.

The following are the implementation details for Part 3:

1) Creation of this class requires torch.distributed to be already initialized, we do this by calling *torch.distributed.init_process_group().* This blocks until all processes have joined.
2) The model is broadcasted at DDP construction time instead of in every forward pass, which helps to speed up training. All processes create a ProcessGroup that enables them to participate in collective communication operations such as AllReduce.
3) The *torch.nn.parallel.DistributedDataParallel()* builds on this functionality to provide synchronous distributed training as a wrapper around any PyTorch model.
4) In this experiment, we use a VGG model as the local model, wrap it with DDP, and then run forward pass, backward pass, and an optimizer step on the DDP model. After that, parameters on the local model will be updated, and all models on different processes should be exactly the same.
5) We use torch.manual_seed() to seed the Random Number Generator for all devices
6) We disabled the benchmarking feature with torch.backends.cudnn.benchmark = False which causes cuDNN to deterministically select an algorithm, possibly at the cost of reduced performance.

Results:

Part 1: Training VGG-11 on Cifar10

Average loss for iterations 1 to 39 : `4.966`
Average time for iterations 1 to 39 : `3.116s`
Average loss on Test dataset after 1 epoch: `2.2838`
Accuracy on Test Dataset after 1 epoch : `1272/10000 (13%)`

```
(base) Chahak@node0:~$ python main1.py
Files already downloaded and verified
Files already downloaded and verified
Train Epoch: 0 [4864/50000 (10%)]       Loss: 6.917100
Train Epoch: 0 [9984/50000 (20%)]       Loss: 2.886030
Train Epoch: 0 [15104/50000 (30%)]      Loss: 2.425694
Train Epoch: 0 [20224/50000 (40%)]      Loss: 2.338928
Train Epoch: 0 [25344/50000 (51%)]      Loss: 2.306177
Train Epoch: 0 [30464/50000 (61%)]      Loss: 2.317816
Train Epoch: 0 [35584/50000 (71%)]      Loss: 2.296219
Train Epoch: 0 [40704/50000 (81%)]      Loss: 2.286434
Train Epoch: 0 [45824/50000 (91%)]      Loss: 2.288492
Finished Training
Test set: Average loss: 2.2838, Accuracy: 1272/10000 (13%)
```

Fig: Average loss after every 20 iterations

```
(base) Chahak@node0:~$ python main1.py
Files already downloaded and verified
Files already downloaded and verified
start time is  1634088895969
end time is  1634088964062
average time is  1745.974358974359
average loss is  4.966151072428777
Finished Training
Test set: Average loss: 2.2838, Accuracy: 1272/10000 (13%)
```

Fig: Screenshot of average time, loss and accuracy for single node training

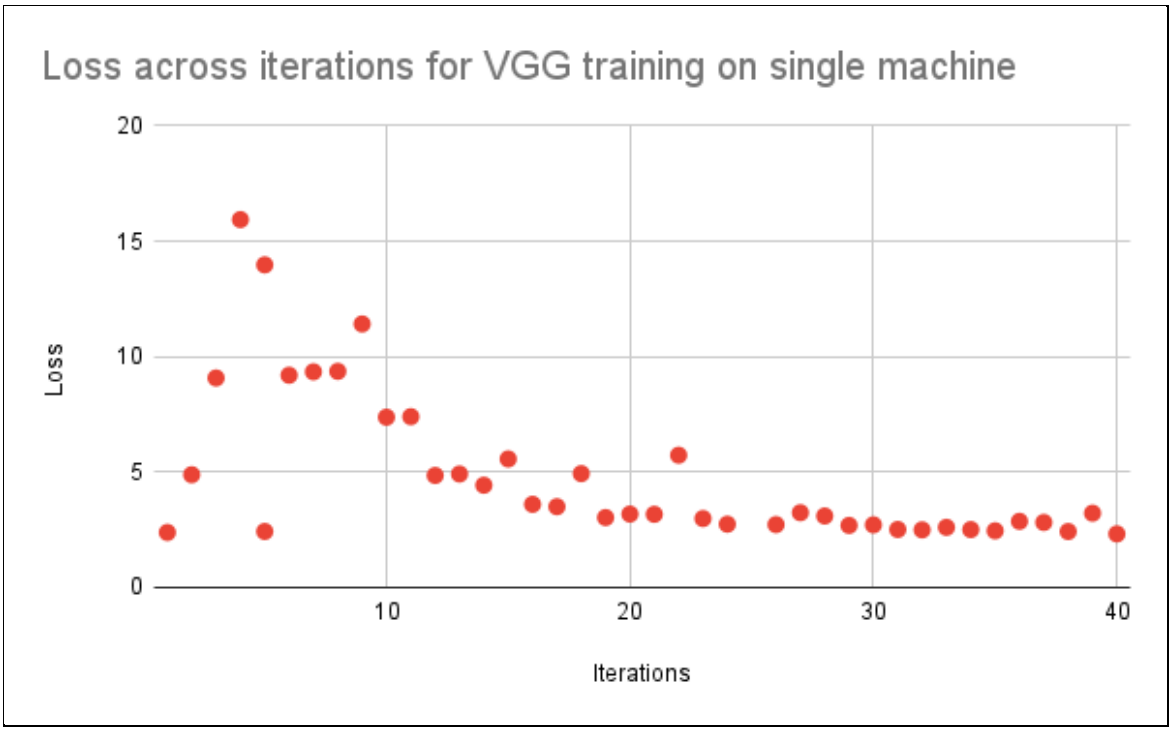


Fig : Change in loss values across iterations (1 to 39) on Node 0

Part 2a: Distributed Data Parallel Training using Gather & Scatter Collectives

Average loss for iterations 1 to 39 : `5.845`
Average time for iterations 1 to 39 : `1.32 seconds`
Average loss on Test dataset after 1 epoch: `2.3036`
Accuracy on Test Dataset after 1 epoch : `1132/10000 (11%)`
Average time for forward pass for iterations 1 to 39 : `0.186229525 seconds`
Average time for backward pass for iterations 1 to 39 : `0.299192575 seconds`
Average time for optimization for iterations 1 to 39 : `0.028162525 seconds`

```
Running Time for 40 iterations excluding first: 51.657107 seconds
Average time per iteration over first 40 iterations excluding first: 1.3245412051282053 seconds
Average loss over 40 iterations excluding first:5.841533129031841
Test set: Average loss: 2.3036, Accuracy: 1132/10000 (11%)
```

Fig: Metrics for DDP using gather-scatter

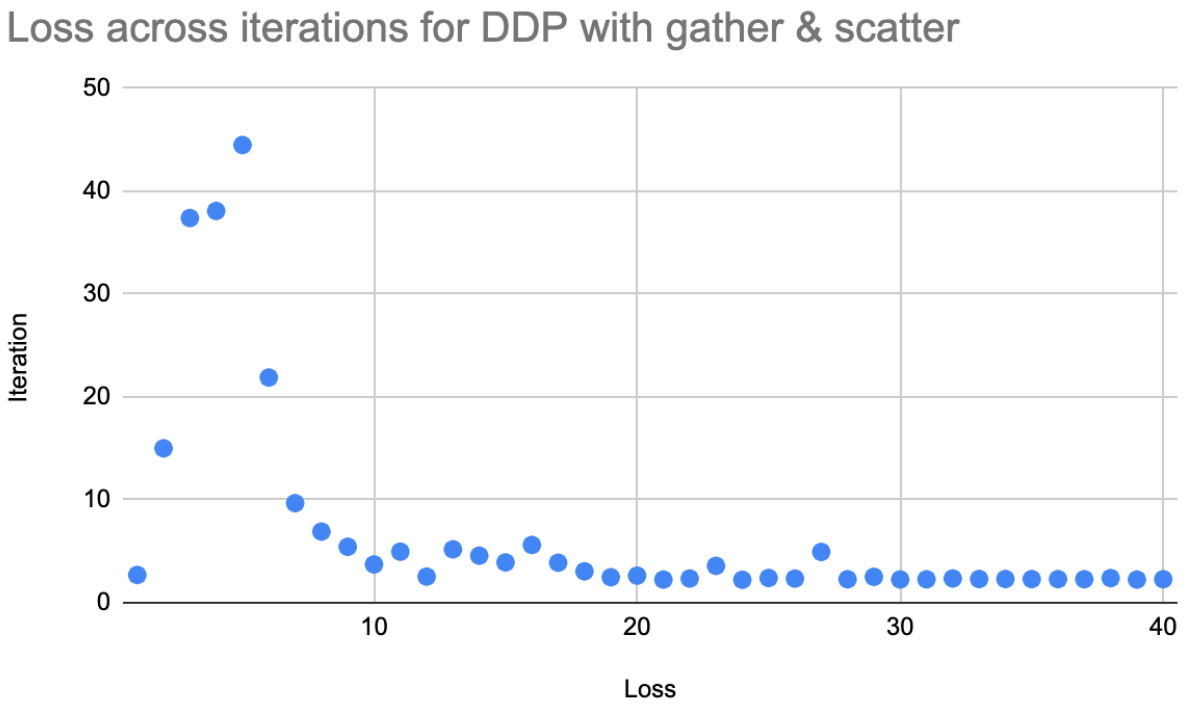


Fig: Loss values across iterations for DDP with gather & scatter

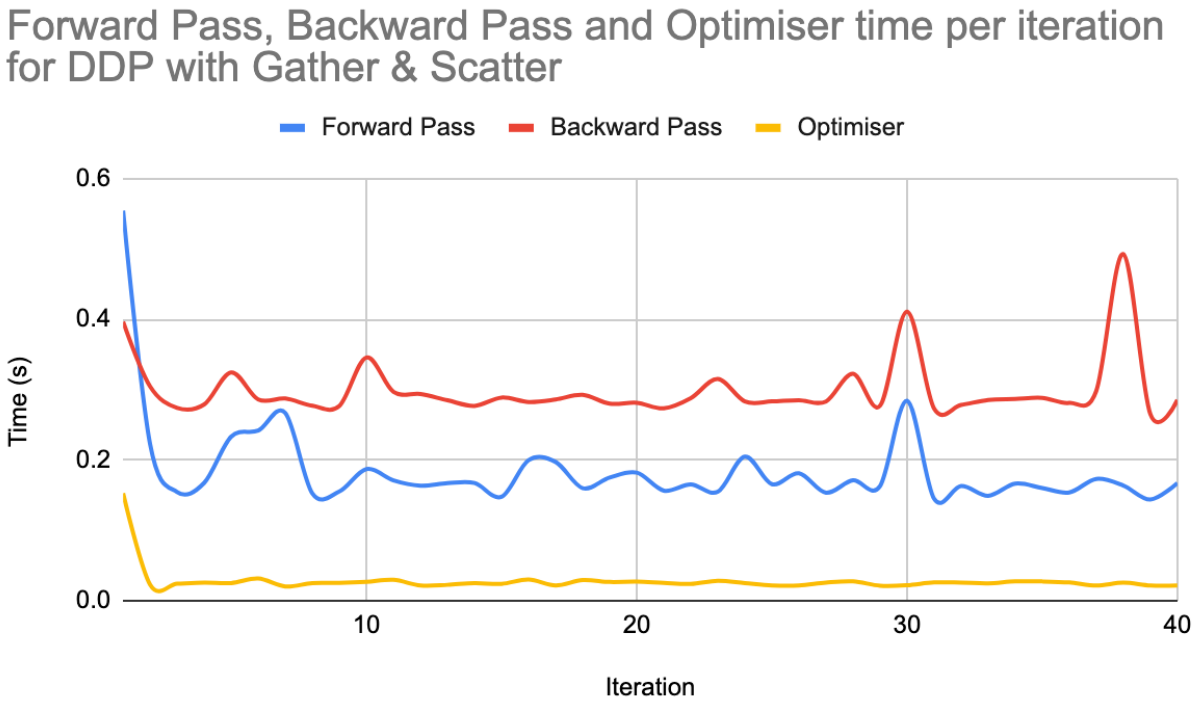
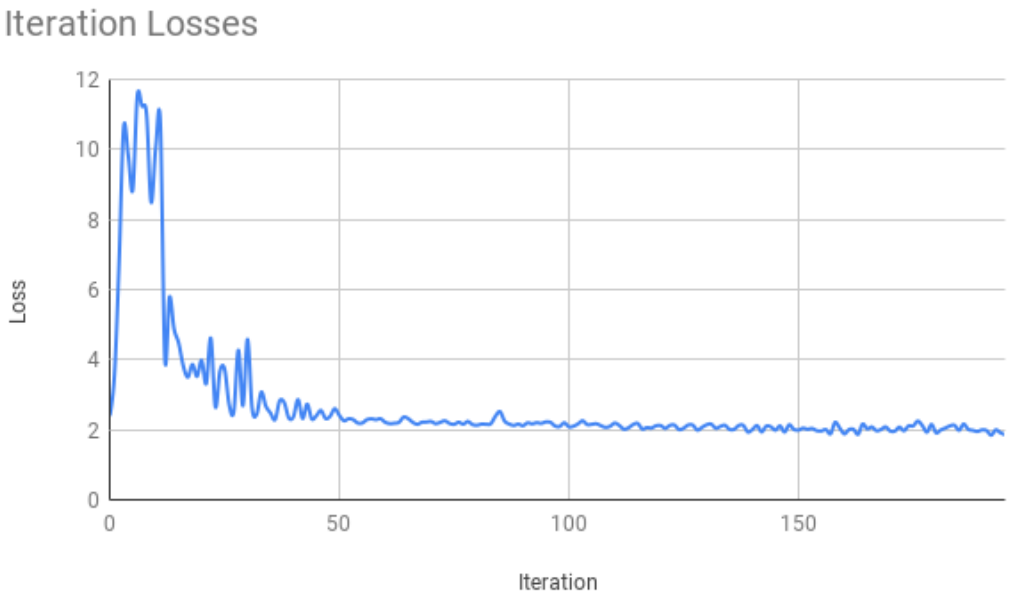
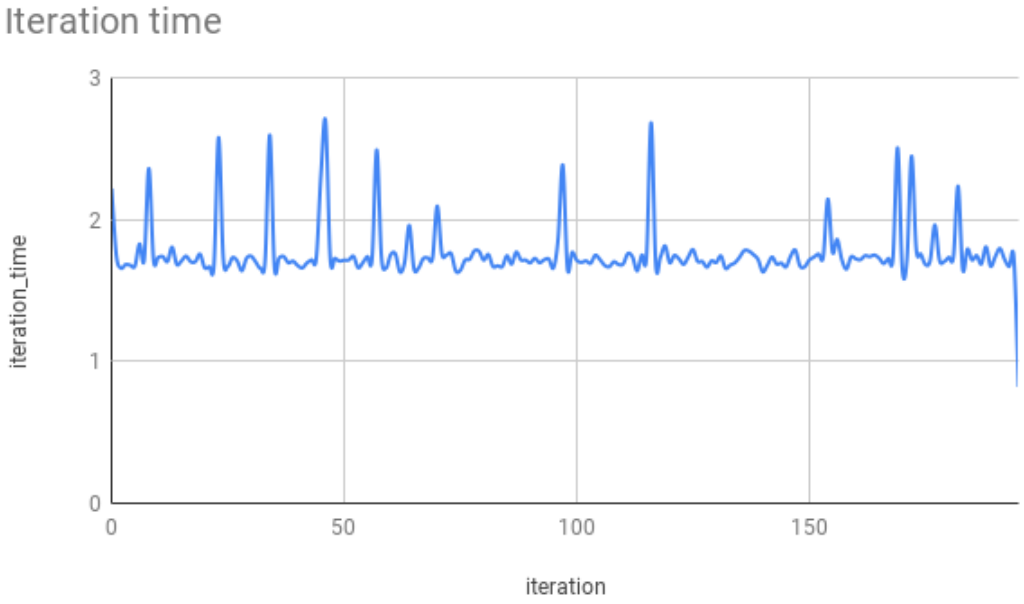


Fig: Time per iteration for Forward Pass, Backward Pass and Optimiser steps

Part 2b: Distributed Data Parallel Training using allreduce



Average loss for iterations 1 to 39 : 5.023950
Average loss on Test dataset after 1 epoch: 2.72324
Accuracy on Test Dataset after 1 epoch : 2237/10000 (22%)



Average time for iterations 1 to 39: 1.76793 seconds

Average time for forward pass for iterations 1 to 39 : 0.44717 seconds
Average time for backward pass for iterations 1 to 39 : 0.86920 seconds
Average time for allreduce pass for iterations 1 to 39 : 0.41320 seconds
Average time for optimization for iterations 1 to 39 : 0.0383 seconds

Part 3: Distributed Data Parallel Training using Built in Module

Average loss for iterations 1 to 39 across all nodes:
Node 0: **5.295**
Node 1: **4.417**
Node 2: **4.721**
Node 3: **4.584**

Average time for iterations 1 to 39 : 0.735s
Average loss on Test dataset : 2.1761
Accuracy on Test Dataset for all nodes after 1 epoch : **1536 / 10000 (15%)**

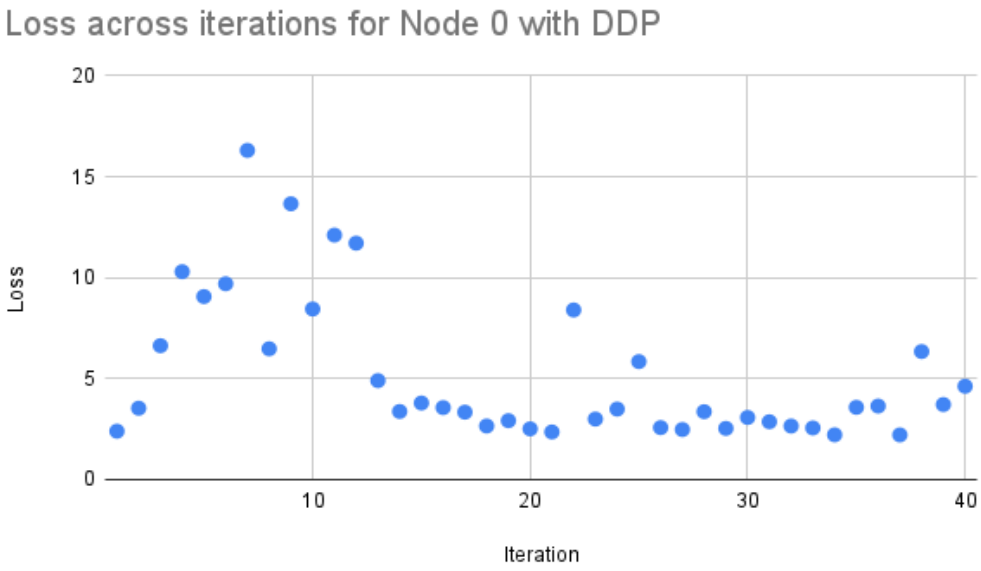


Fig: Change in loss values across iterations (1 to 39) on Node 0

Screenshot for loss, average time and accuracy on all Nodes:

```
(base) Chahak@node0:~$ python mainDDP.py --master-ip 10.10.1.1
--num-nodes 4 --rank 0
64
Files already downloaded and verified
Files already downloaded and verified
start time is  1634109137415
end time is  1634109166068
average time is  734.6923076923077
average loss is  5.294978979306343
Finished Training
Test set: Average loss: 2.1761, Accuracy: 1536/10000 (15%)
```

```
(base) Chahak@node1:~$ python mainDDP.py --master-ip 10.10.
1.1 --num-nodes 4 --rank 1
Files already downloaded and verified
Files already downloaded and verified
start time is  1634109137407
end time is  1634109166060
average time is  734.6923076923077
average loss is  4.417418614411965
Finished Training
Test set: Average loss: 2.1761, Accuracy: 1536/10000 (15%)
```

```
(base) Chahak@node2:~$ python mainDDP.py --master-ip 10.10.
1.1 --num-nodes 4 --rank 2
Files already downloaded and verified
Files already downloaded and verified
start time is  1634109137377
end time is  1634109166051
average time is  735.2307692307693
average loss is  4.721644878387451
Finished Training
Test set: Average loss: 2.1761, Accuracy: 1536/10000 (15%)
```

```
(base) Chahak@node3:~$ python mainDDP.py --master-ip 10.10.
1.1 --num-nodes 4 --rank 3
Files already downloaded and verified
Files already downloaded and verified
start time is  1634109137382
end time is  1634109166052
average time is  735.1282051282051
average loss is  4.584576331652128
Finished Training
Test set: Average loss: 2.1761, Accuracy: 1536/10000 (15%)
```

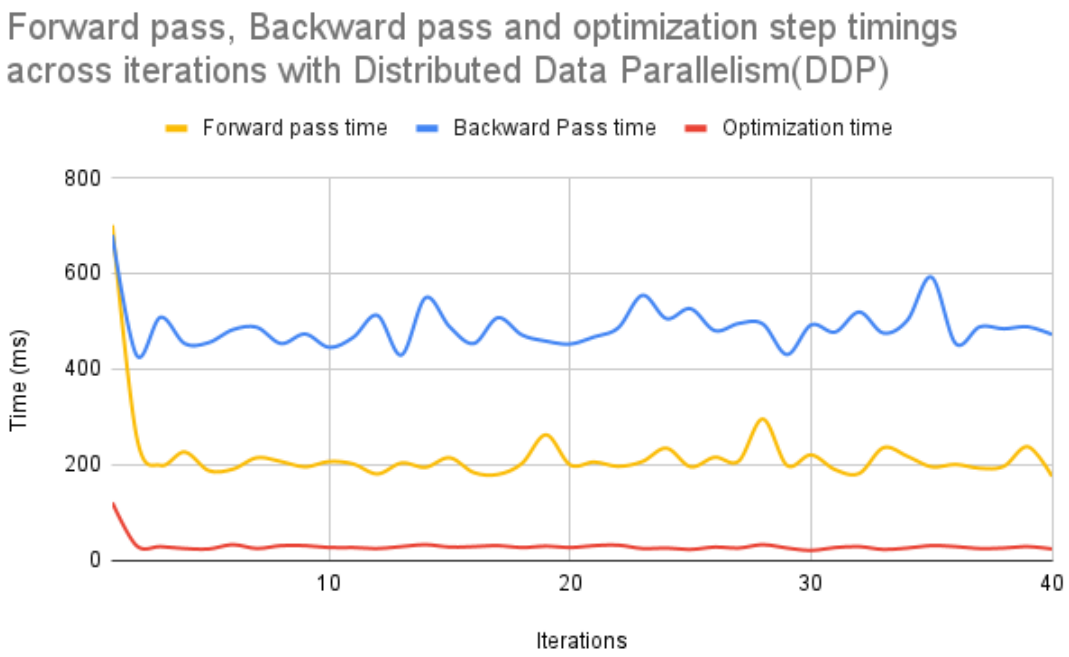


Fig: Plot showing the time taken for different stages in training process (forward pass, backward pass and optimization steps)

Inferences:

1) We observe a drop in training time with DDP using gather & scatter from ~114s in Task 1, to ~47s in Task 2a. This can be attributed to splitting the data to 4 workers, and overlapping computation with communication while each node is performing work.
2) Between tasks 2a, and 2b, we see that the average time taken to train was quite close. Ideally we would expect scatter & gather to perform better with a large number of nodes because only the master (rank 0) would send the averaged gradients to other nodes upon receiving individual gradients from all workers. But since our topology has only 4 nodes, the differences are quite small.
3) We observe a drop in training time with DDP (built-in module) from ~47s in Task 2 to ~29s for Task 3. This is in accordance with the documentation, as, in DDP, each process performs a complete optimization step with each iteration. Since the gradients have already been gathered together and averaged across processes, no parameter broadcast step is needed, reducing latency due to time spent transferring tensors between nodes.
4) DDP ensures mathematical equivalence, therefore we expect to harvest the same result model on Task 3 as if all training had been performed locally like in Task 1. To guarantee mathematical equivalence, all replicas start from the same initial values for model parameters and synchronize gradients to keep parameters consistent after each iteration.
5) We started with the same seed values on all nodes. We observe similar accuracy values on the test data for Task 1 and 3 after training for 1 epoch, 13% and 15% respectively. We also observed that the gradients after the backward pass were the same across all nodes for DDP.
6) We also see considerable improvement in the time taken to train the model using built in DDP versus using the gather-scatter & all reduce collectives. This can be explained by the fact that DDP includes built-in optimizations like gradient bucketing, which overlaps communication with computation unlike gather-scatter and allreduce.
7) The results demonstrate that the backward pass is the most time-consuming step with PyTorch DDP training. It takes about twice as long as the forward pass, which is in agreement with the PyTorch/PipeDream papers.

Scalability

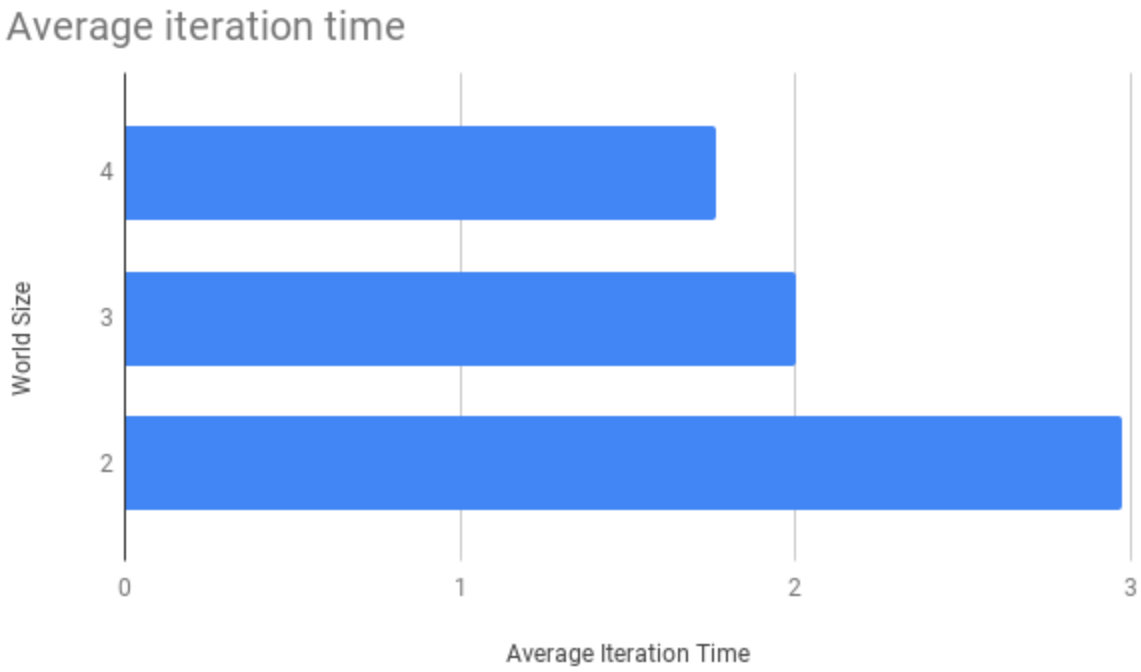
Below is a table that summarizes the average loss, the average time taken and the accuracy that each of the parts gave. As the table shows, the machine learning model learns faster when performing the operations on multiple nodes, using data-parallel, and distributed data-parallel training paradigms. The reason for this being that distributed data-parallel attempts to overlap communication with computation, hence the work done by each of the nodes is more.

	Avg loss (iterations 1->39)	Avg time (iterations(1->39)	Avg loss (1 epoch)	Accuracy on Test Dataset (1 epoch)
Single Node	4.966	3.116s	2.284	13%
DDP with Gather & Scatter	5.841	1.324s	2.304	11%
DDP with AllReduce	5.02390	1.767s	2.723	22%
DDP with built-in module	4.584	0.735s	2.176	15%

Part 2b: Distributed Data Parallel Training using allreduce

The below table shows the average iteration time when world size is varied between 2 and 4, for DDP with AllReduce

World Size	Average Iteration time (Iterations 1-40)	Average iteration time (1 Epoch)
4	1.767 seconds	1.762 seconds
3	1.984 seconds	2.003 seconds
2	3.010 seconds	2.974 seconds



We observe a ~30% reduction in average iteration time when moving from two to three worker nodes, and a ~12% reduction in average iteration time when moving from three to four worker nodes. From our observations, it looks like the allreduce based training approach is not infinitely scalable and has diminishing returns as more workers are added. This might be due to the additional communication overheads introduced due to adding a new worker.

Member Contributions

1) Chahak Tharani worked on Part 1 where she wrote the functionality for training the network on a single node.
2) Deepti Rajagopal worked on Part 2a where she implemented DDP using gather and scatter collectives.
3) Akul Gupta worked on Part 2b where he implemented DDP using AllReduce collectives.
4) Chahak Tharani worked on Part 3 where she employed the DDP model over the basic training model and compared the performance of single machine vs distributed multi node training mechanism.
5) All members worked on deducing inferences, comparing different models, and gathering results.
6) All members contributed equally to this report.

References

[1] Pytorch distributed: experiences on accelerating data parallel training towards a unified architecture for in-rdbms analytics - Shen Li and Yanli Zhao and Rohan Varma and Omkar Salpekar and Pieter Noordhuis and Teng Li and Adam Paszke and Jeff Smith and Brian Vaughan and Pritam Damania and Soumith Chintala, arXiv 2020

[2] PyTorch Distributed docs -> https://pytorch.org/docs/stable/distributed.html

[3] Assignment 2 - https://pages.cs.wisc.edu/~shivaram/cs744-fa21/assignment2.html