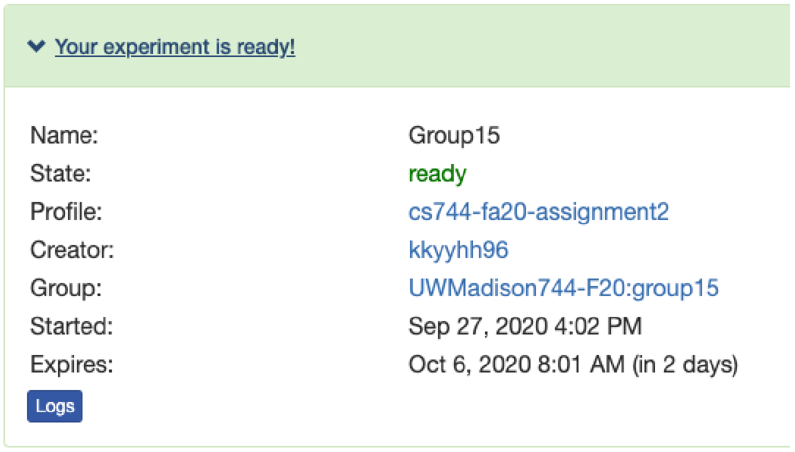
CS 744 Assignment 2

Yuhao Kang (yuhao.kang@wisc.edu), Jinmeng Rao (jinmeng.rao@wisc.edu)

Part 0 - Environment Setup

We follow the instructions on the course website, setup the Anaconda python environment and PyTorch deep learning package on the four-node cluster machines.

Here is the cluster machine information (Figure 1):



Your experiment is ready!

Name:	Group15
State:	ready
Profile:	cs744-fa20-assignment2
Creator:	kkyyhh96
Group:	UWMadison744-F20:group15
Started:	Sep 27, 2020 4:02 PM
Expires:	Oct 6, 2020 8:01 AM (in 2 days)

Figure 1 Machine Information

To do so, we first download the latest Anaconda 3.8 from the official website: https://www.anaconda.com/products/individual. Then, after installing the Anaconda, we download PyTorch and installed successfully by running:

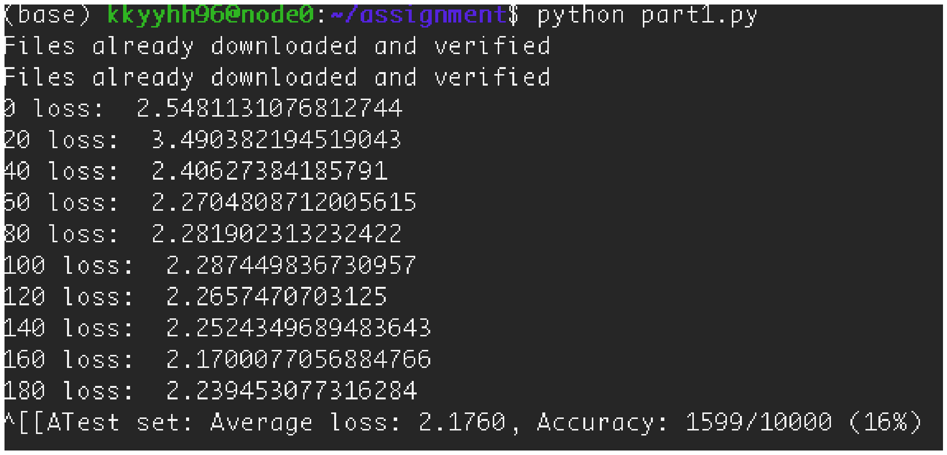
pip install torch==1.4.0+cpu torchvision==0.5.0+cpu -f https://download.pytorch.org/whl/torch_stable.html

By running *torch.distributed.is_available()*, the distributed package is available and has been installed.

In addition, some parameters settings are illustrated here. In order to have consistent results, the random seed of both PyTorch and NumPy are assigned as 5000, and the master node port is assigned as 29501.

Part 1 - Training VGG-11 on Cifar10

In this part, we completed the based training scripts. We finished the standard training loop of forward and backward pass, loss computation and optimizer in main.py, and printed loss every 20 iterations. The printed loss value and the calculated test accuracy is shown in Figure 2. As is shown in the Figure 2, the final average loss is 2.1760 and the test accuracy is 16%, which are all expected.



```
(base) kkyyhh96@node0:~/assignment$ python part1.py
Files already downloaded and verified
Files already downloaded and verified
0 loss:  2.5481131076812744
20 loss:  3.490382194519043
40 loss:  2.40627384185791
60 loss:  2.2704808712005615
80 loss:  2.281902313232422
100 loss:  2.287449836730957
120 loss:  2.2657470703125
140 loss:  2.2524349689483643
160 loss:  2.1700077056884766
180 loss:  2.239453077316284
^[[ATest set: Average loss: 2.1760, Accuracy: 1599/10000 (16%)
```

Figure 2 Loss value printed every 20 iterations, and test accuracy (part1)

As required, we also recorded the average running time per iterations. We run 10 iterations, ignored the first iterations, and calculated the average running time of the remaining 9 iterations. The average time per iteration is: **2.4656721** s

Part 2 - Distributed Data Parallel Training

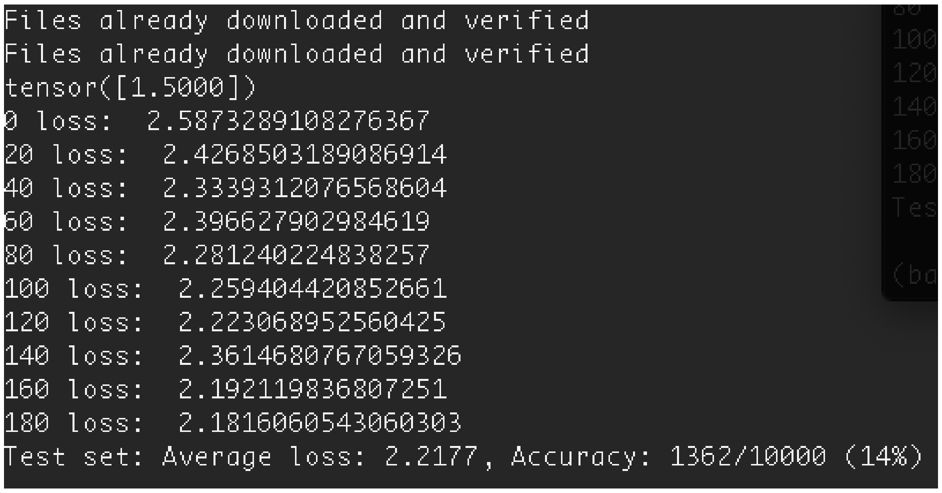
Part 2a - Sync gradient with gather and scatter call using Gloo backend

In part 2a, two approaches are used in this section. Apart from the standard solution which uses gather/scatter functions, we also finished the task using isend/irecv, and we decided to provide both solutions and codes in our assignment deliverables as extra findings.

(1) Gather/scatter call

The standard solution is based on *torch.distributed.gather* and *torch.distributed.scatter*. We first gather 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. Please note that the batch size is changed to 64 as there are 4 machines in total. So that 64 x 4 = 256 which is equal to the batch size in part1.

The training process went as expected, and the printed loss value and the calculated test accuracy is shown in Figure 3. As is shown in the Figure, the final average loss is 2.2177 and the test accuracy is 14%, which are slightly different from part1, but are still expected. Also, the changing trend is very similar to the results in part1.



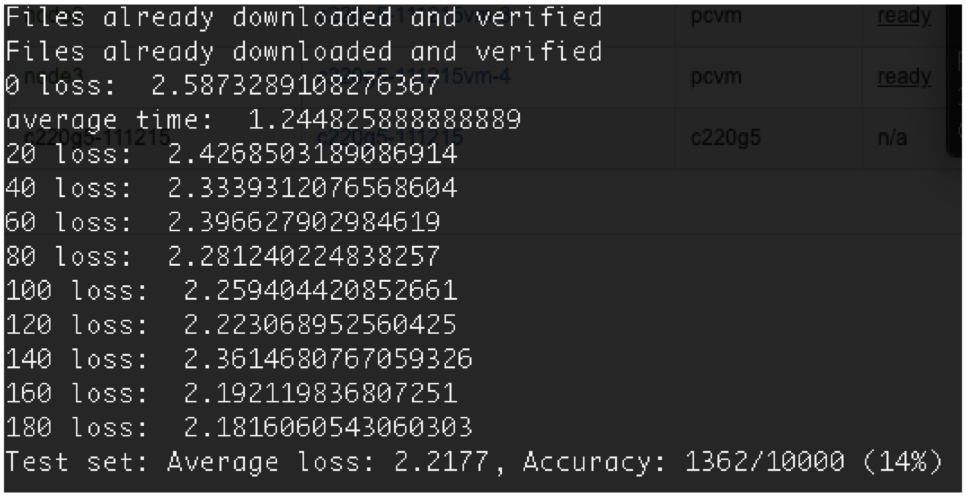
```
Files already downloaded and verified
Files already downloaded and verified
tensor([1.5000])
0 loss:  2.5873289108276367
20 loss:  2.4268503189086914
40 loss:  2.3339312076568604
60 loss:  2.396627902984619
80 loss:  2.281240224838257
100 loss:  2.259404420852661
120 loss:  2.223068952560425
140 loss:  2.3614680767059326
160 loss:  2.192119836807251
180 loss:  2.1816060543060303
Test set: Average loss: 2.2177, Accuracy: 1362/10000 (14%)
```

Figure 3 Loss value printed every 20 iterations, and test accuracy (part2a, gather/scatter)

As required, we also recorded the average running time per iterations. We run 10 iterations, ignored the first iterations, and calculated the average running time of the remaining 9 iterations. The average time per iteration is: **1.6191249 s**

*(2) isend/irecv call

Apart from the gather/scatter solution, we also provide an extra solution that uses *isend/irecv* call. The basic logic is the same: For nodes 1-3, we first use *isend* call to send all gradients from *model.parameters()* to node 0, and use *irecv* call to receive the mean gradients from node 0. For node 0, we first use *irecv* call to receive the gradients from all the other codes, and use isend to send the calculated mean gradients to all the other nodes. The training process went as expected, and the printed loss value and the calculated test accuracy is shown in Figure 4. As is shown in the Figure, the final average loss is 2.2177 and the test accuracy is 14%, which are expected.



```
Files already downloaded and verified
Files already downloaded and verified
0 loss:  2.5873289108276367
average time:  1.244825888888889
20 loss:  2.4268503189086914
40 loss:  2.3339312076568604
60 loss:  2.396627902984619
80 loss:  2.281240224838257
100 loss:  2.259404420852661
120 loss:  2.223068952560425
140 loss:  2.3614680767059326
160 loss:  2.192119836807251
180 loss:  2.1816060543060303
Test set: Average loss: 2.2177, Accuracy: 1362/10000 (14%)
```

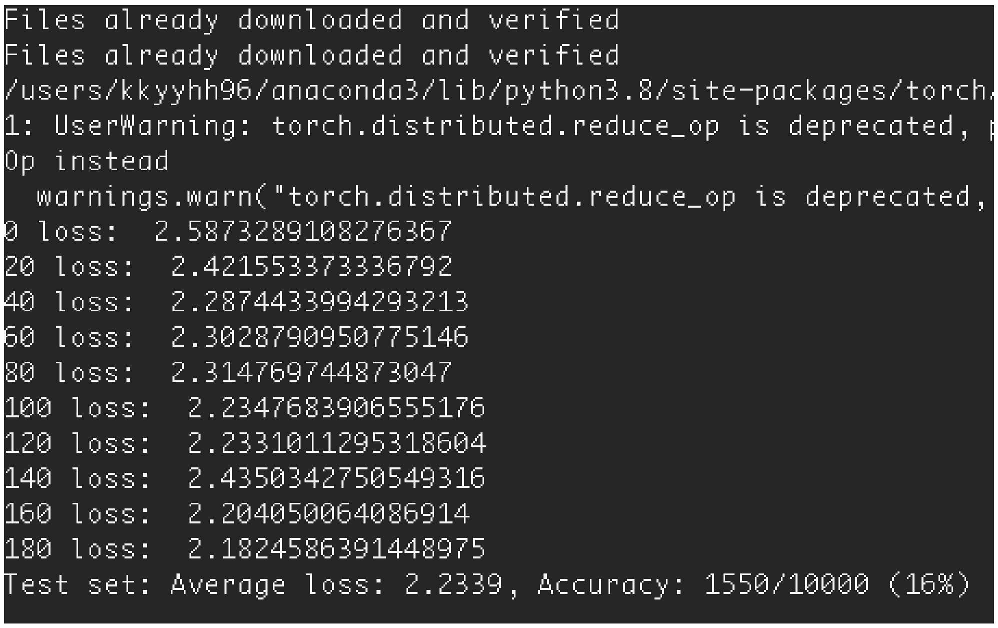
Figure 4 Loss value printed every 20 iterations, and test accuracy (part2a, isend/irecv)

As required, we also recorded the average running time per iterations. We run 10 iterations, ignored the first iterations, and calculated the average running time of the remaining 9 iterations. The average time per iteration is: **1.2372376 s**

Part 2b - Sync gradient with allreduce using Gloo backend

In this part, we use *allreduce* to replace the gather/scatter function to sync gradients among different nodes. We all-reduce all the gradients for each layer using *dist.reduce_op.SUM* mode and average them using the number of nodes, and sync them to all nodes. Note that in the code, we first divide all the gradients by the number of nodes (which is 4), then we all-reduce them using SUM mode. This is equivalent to first summing them then averaging them.

The training process went as expected, and the printed loss value and the calculated test accuracy is shown in Figure 5. As is shown in the Figure 5, the final average loss is 2.2339 and the test accuracy is 16%, which are expected.



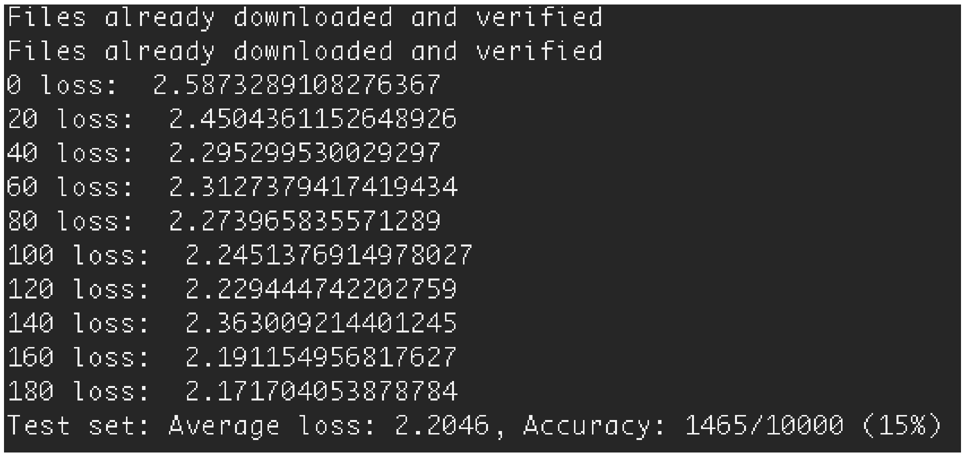
```
Files already downloaded and verified
Files already downloaded and verified
/users/kkyyhh96/anaconda3/lib/python3.8/site-packages/torch/
1: UserWarning: torch.distributed.reduce_op is deprecated, p
Op instead
  warnings.warn("torch.distributed.reduce_op is deprecated,
0 loss:  2.5873289108276367
20 loss:  2.421553373336792
40 loss:  2.2874433994293213
60 loss:  2.3028790950775146
80 loss:  2.314769744873047
100 loss:  2.2347683906555176
120 loss:  2.2331011295318604
140 loss:  2.4350342750549316
160 loss:  2.204050064086914
180 loss:  2.1824586391448975
Test set: Average loss: 2.2339, Accuracy: 1550/10000 (16%)
```

Figure 5 Loss value printed every 20 iterations, and test accuracy (part2b)

As required, we also recorded the average running time per iterations. We run 10 iterations, ignored the first iterations, and calculated the average running time of the remaining 9 iterations. The average time per iteration is: **1.128978 s**

Part 3 - Distributed Data Parallel Training using Built in Module

In this part, we used DDP to automatically perform gradient synchronization among all the nodes. We wrapped the VGG model using the DistributedDataParallel API, and the gradient synchronization is achieved. The training process went as expected, and the printed loss value and the calculated test accuracy is shown in Figure 6. As is shown in the Figure, the final average loss is 2.2046 and the test accuracy is 15%, which are expected.



```
Files already downloaded and verified
Files already downloaded and verified
0 loss:  2.5873289108276367
20 loss:  2.4504361152648926
40 loss:  2.295299530029297
60 loss:  2.3127379417419434
80 loss:  2.273965835571289
100 loss:  2.2451376914978027
120 loss:  2.229444742202759
140 loss:  2.363009214401245
160 loss:  2.191154956817627
180 loss:  2.171704053878784
Test set: Average loss: 2.2046, Accuracy: 1465/10000 (15%)
```

Figure 6 Loss value printed every 20 iterations, and test accuracy (part3)

As required, we also recorded the average running time per iterations. We run 10 iterations, ignored the first iterations, and calculated the average running time of the remaining 9 iterations. The average time per iteration is: **0.8620212 s**

Reason about the difference among different setups

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.

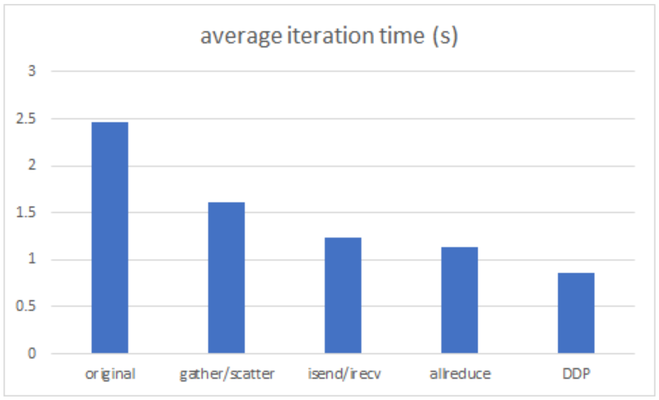


Figure 7 Average iteration time of different setup methods

Here we concluded and compared the average iteration time of different setup methods (Figure 7). As we can see, the original method (basic training script without distributed setup) takes the longest average iteration time (2.4656721 s). The time cost decreases as we switch to gather/scatter, isend/irecv, AllReduce, and then DDP. DDP takes the shortest average iteration time (0.8620212 s).

This suggests that, compared with the original singal-machine training setup, the distributed setup can significantly improve the training speed since it uses more nodes and thus can do the training task parallelly (e.g., given total batch size as 256, distributed setup can parallelly process four 64-batches on 4 nodes at the same time). Also, compared with the low-level calls such as gather/scatter and isend/irecv, the relatively high-level distributed setup API such as AllReduce and DDP (especially DDP, which is the built-in module) are better designed and optimized and thus can better lower the training time. For example, AllReduce in PyTorch is ring-based and thus are more efficient than scatter/gather or isend/irecv. DDP further organizes small gradients into larger buckets and synchronizes each bucket using an AllReduce call in order to reduce the total number of Allreduce used. The optimization greatly decreases the communication cost, thus improving the training speed.

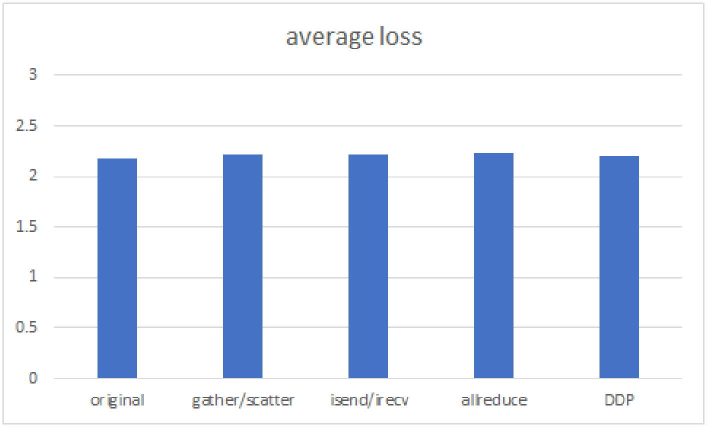


Figure 8 Average loss of different setup methods

Here we concluded and compared the average loss of different setup methods (Figure 8). As we can see, generally, the average loss doesn't have a significance difference among all the setups, which suggests that different distributed setups won’t have a significant impact on the training loss. This is reasonable since (1) the training data we use for all the setups are the same; (2) the models for each setup remain the same (VGG16); and (3) the total epoch, total batchsize, and training parameters remain the same. These are the reasons why the average loss doesn’t change a lot among all setups.

However, even though we set the same PyTorch/Numpy random seed for all setups, there is still slight difference among different setups in the average loss, as well as the loss printed at every 20 iterations. There are probably two reasons: (1) There are Batch

Normalization (BN) layers in the given VGG models defined in model.py, and the parameters in these BN layers are not synchronized among all the nodes during distributed training, which brings uncertainty and randomness to the training and causes the slight difference in loss among all the setups; (2) due to the float computation precision, there might be some minor errors during the training each time, which also adds uncertainty and randomness to the loss computation result.

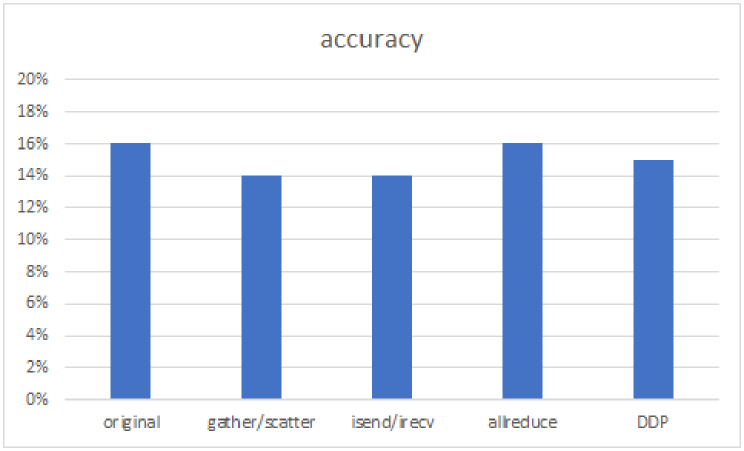


Figure 9 Test accuracy of different setup methods

Here we concluded and compared the test accuracy of different setup methods (Figure 9). As we can see, generally, the test accuracy doesn't have a significance difference among all the setups. However, even though we set the same PyTorch/Numpy random seed for all setups, there is still slight difference among different setups in the test accuracy. The reasons are probably the same as the abovementioned reasons for different average loss. In general, different distributed setups won't have a significant impact on the test accuracy.

Comment on the scalability of distributed machine learning

In this section, we discuss the scalability of distributed machine learning based on our results, especially the difference among different distributed setup in PyTorch.

We compare the average iteration time (including distributed communication latency, if applicable) between the setups with different number of nodes to evaluate the scalability of the method. That is, we run the same distributed program on single machine, 2-node machine, and 4-node machine, respectively, and calculate and compare their average iteration time (also without the start iteration). The comparison results can be seen in Figure 10.

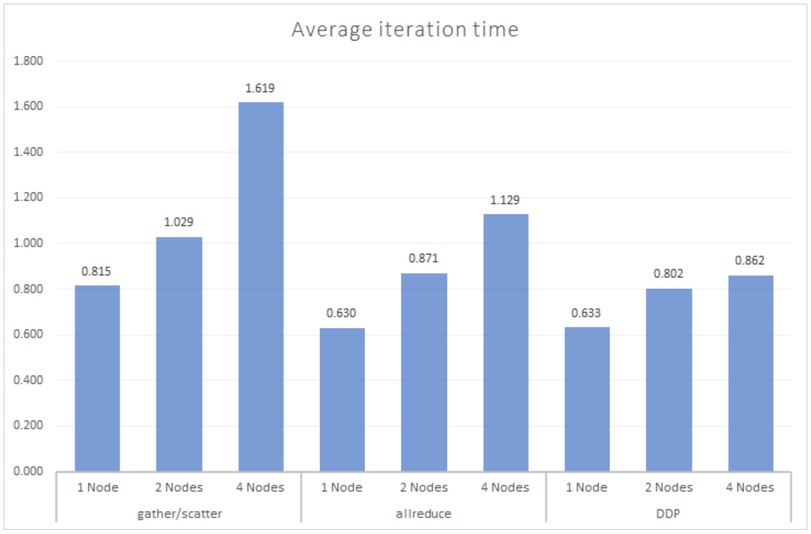


Figure 10 Average iteration time (including latency) among different setups

As can be seen in Figure 10, for each distributed setup, the average iteration time (including latency) increases as the number of nodes increases (close to linear increment). For scatter/gather, it increases from 0.815 (1 node) to 1.619 (4 nodes). This doubled time cost indicates the relatively low scalability of this setup. For DDP, the average iteration time is very similar among all settings with different nodes, and it is also the smallest among all setups in general. This shows that DDP has relatively high scalability. The scalability of allreduce is located between gather/scatter and DDP. This result makes sense because, as we mentioned in the last section, allreduce and DDP are optimized (DDP is even better optimized), which greatly lowers the per iteration latency, thus having higher scalability.

Apart from this finding, we also should know that we need to choose the distributed techniques to use based on the size of the given model and available resources in order to strike a better balance between scalability and training speed. Different models may have different network structure, parameter size, and layer distribution, thus having different best suitable distributed setup.

Piazza participation and problem solving

In this assignment, we read many materials to understand how PyTorch Distributed API works and how to fix the problem we met. Apart from the PyTorch Distributed paper, we also found many materials Professor gave very helpful, which includes (not limited to):

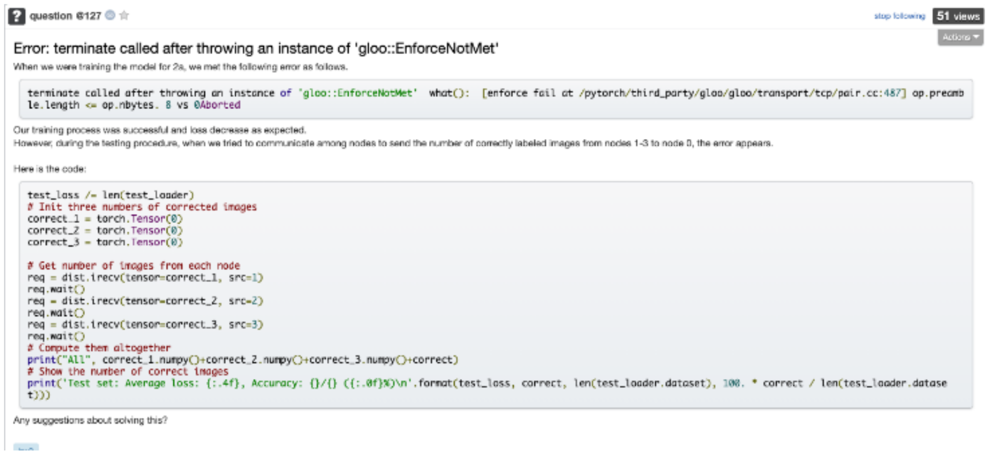
https://pytorch.org/docs/stable/distributed.html#launch-utility

https://pytorch.org/docs/stable/distributed.html#initialization https://github.com/pytorch/examples/tree/master/distributed/ddp

https://pytorch.org/tutorials/intermediate/dist_tuto.html

https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

Also, we met two problems that we cannot solve by ourselves in a short time, so we posted two questions on Piazza and asked for help (Figure 11-13). Prof. Venkataraman provided very useful suggestions to us that helped solved the problems. We greatly appreciate professor's help!



Error: terminate called after throwing an instance of 'gloo::EnforceNotMet'

When we were training the model for 2a, we met the following error as follows.

```
terminate called after throwing an instance of 'gloo::EnforceNotMet'  what():  [enforce fail at /pytorch/third_party/gloo/gloo/transport/tcp/pair.cc:487] op.preamb
le.length <= op.nbytes. 8 vs 0Aborted
```

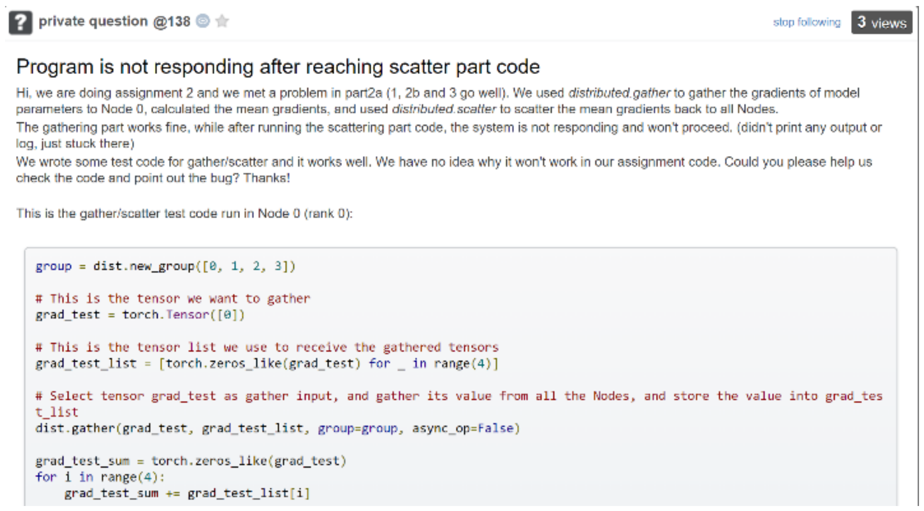
Our training process was successful and loss decrease as expected.
However, during the testing procedure, when we tried to communicate among nodes to send the number of correctly labeled images from nodes 1-3 to node 0, the error appears.

Here is the code:

```
test_loss /= len(test_loader)
# Init three numbers of corrected images
correct_1 = torch.Tensor(0)
correct_2 = torch.Tensor(0)
correct_3 = torch.Tensor(0)

# Get number of images from each node
req = dist.irecv(tensor=correct_1, src=1)
req.wait()
req = dist.irecv(tensor=correct_2, src=2)
req.wait()
req = dist.irecv(tensor=correct_3, src=3)
req.wait()
# Compute them altogether
print("All", correct_1.numpy()+correct_2.numpy()+correct_3.numpy()+correct)
# Show the number of correct images
print('Test set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(test_loss, correct, len(test_loader.dataset), 100. * correct / len(test_loader.datase
t)))
```

Any suggestions about solving this?

Figure 11 Piazza screenshot 1



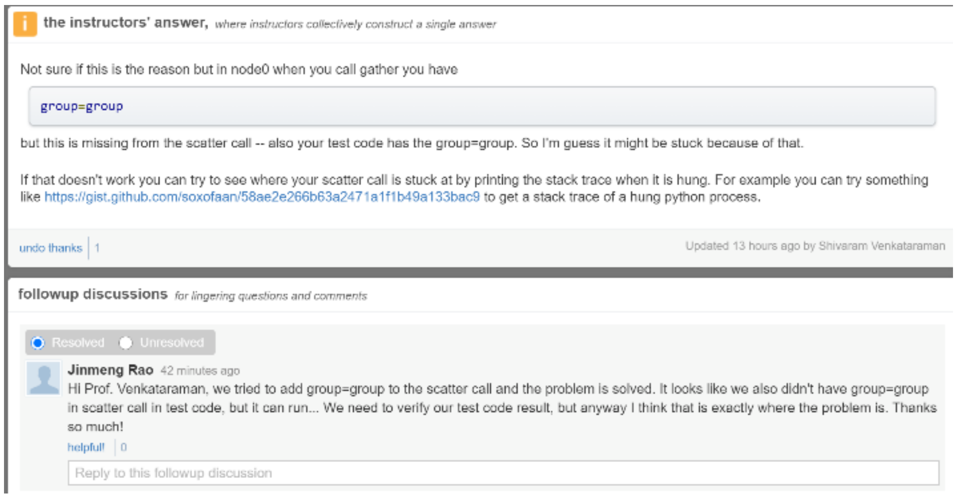
Program is not responding after reaching scatter part code

Hi, we are doing assignment 2 and we met a problem in part2a (1, 2b and 3 go well). We used *distributed.gather* to gather the gradients of model parameters to Node 0, calculated the mean gradients, and used *distributed.scatter* to scatter the mean gradients back to all Nodes.
The gathering part works fine, while after running the scattering part code, the system is not responding and won't proceed. (didn't print any output or log, just stuck there)
We wrote some test code for gather/scatter and it works well. We have no idea why it won't work in our assignment code. Could you please help us check the code and point out the bug? Thanks!

This is the gather/scatter test code run in Node 0 (rank 0):

```
group = dist.new_group([0, 1, 2, 3])

# This is the tensor we want to gather
grad_test = torch.Tensor([0])

# This is the tensor list we use to receive the gathered tensors
grad_test_list = [torch.zeros_like(grad_test) for _ in range(4)]

# Select tensor grad_test as gather input, and gather its value from all the Nodes, and store the value into grad_tes
t_list
dist.gather(grad_test, grad_test_list, group=group, async_op=False)

grad_test_sum = torch.zeros_like(grad_test)
for i in range(4):
    grad_test_sum += grad_test_list[i]
```

Figure 12 Piazza screenshot 2


the instructors' answer, *where instructors collectively construct a single answer*

Not sure if this is the reason but in node0 when you call gather you have

```
group=group
```

but this is missing from the scatter call -- also your test code has the group=group. So I'm guess it might be stuck because of that.

If that doesn't work you can try to see where your scatter call is stuck at by printing the stack trace when it is hung. For example you can try something like https://gist.github.com/soxofaan/58ae2e266b63a2471a1f1b49a133bac9 to get a stack trace of a hung python process.

undo thanks | 1
Updated 13 hours ago by Shivaram Venkataraman
followup discussions *for lingering questions and comments*
Resolved Unresolved

Jinmeng Rao 42 minutes ago
Hi Prof. Venkataraman, we tried to add group=group to the scatter call and the problem is solved. It looks like we also didn't have group=group in scatter call in test code, but it can run... We need to verify our test code result, but anyway I think that is exactly where the problem is. Thanks so much!

Figure 13 Piazza screenshot 3

Contributions

Yuhao Kang and Jinmeng Rao worked on this assignment collaboratively and contributed equally.

Yuhao Kang mainly worked on Part 0, Part1, Part2a-gather/scatter, Part2b, and script wrap-up.

Jinmeng Rao mainly worked on Part2a-isend/irecv, Part3, result discussion and report drafting.