SREYASHI SAHA **Matriculation No: 1747271** **Group 1**

EXERCISE 1:

```
from mpi4py import MPI
import numpy as np
import torch
import torch.nn as nn
import torchvision.datasets as datasets
import torch.optim as optim
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import torch.nn.functional as F
from torch.utils.data import random_split
```

Importing all the libraries required to implement MNIST architecture on MNIST data set. This time we will use Pytorch in MPI, i.e. our calculations will be done in parallel.

```
class Mnist(nn.Module):
  def __init__(self):
    super(Mnist, self).__init__()

    self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding = 1)
    self.pool1 = nn.MaxPool2d(kernel_size=2, stride = 2, padding = 1)
    self.conv2 = nn.Conv2d(in_channels=16, out_channels=64, kernel_size=5, stride=1, padding = 0)
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride = 2)
    self.pool3 = nn.MaxPool2d(kernel_size=5, stride = 1)
    self.fc1 = nn.Linear(in_features=64, out_features=32)
    self.fc2 = nn.Linear(in_features=32, out_features=10)
    self.dp1 = nn.Dropout(0.5)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool1(x)
    x = F.relu(self.conv2(x))
    x = self.pool2(x)
    x = self.dp1(x)
    x = self.pool3(x)
    x = x.reshape(x.shape[0], -1)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

I have defined my architecture as asked in the question. The CNN architecture has eight layers. Two convolutional layers, three subsampling layers, and two fully linked layers make up the layer composition.
The input layer, which is the initial layer, is typically not thought of as part of the network because nothing is learned there. The following layer receives photos that are 32x32 in size since the input layer is designed to accept images of that size. The images in the MNIST collection are 28x28 sized images, which are padded to bring the MNIST images' dimensions into compliance with the input layer's specifications.

The Formula used to get the output dimensions after each layer is:
(W−F+2P)/S+1 ,
where W is the input height/width, F is the filter/kernel size, P is the padding, and S is the stride.

Using the current weights, I get the predictions for each batch of data in the train function. This is known as the Forward Pass. I then figure out the loss function's value. I follow this with a backward pass, where the weights are changed in accordance with the loss. This is known as the "Learning Phase". The model is in training mode (model.train()) for the training phase, and I have also zeroed out the gradients for each batch. Additionally, I find out the running loss throughout the training phase. I then return the model which I'll be implementing, in this case it is Moist model, along with the optimizer used i.e. Sgd and the loss obtained after each epoch from the training dataset.

```
def train(model, criterion, optimizer, dataloader, dataset_size, training_loss, train_accuracy):
    running_loss = 0.0
    no_of_correct_predictions = no_of_samples = 0


    for data in dataloader:
        input_x, labels_y  = data
        batch_size = input_x.shape[0]

        input_x = input_x.to(device)
        labels_y = labels_y.to(device)

        optimizer.zero_grad()
        outputs_y_hat = model(input_x)
        loss = criterion(outputs_y_hat, labels_y)

        loss.backward()
        optimizer.step()

        running_loss += loss.data * batch_size

        _, preds = outputs_y_hat.max(1)
        no_of_correct_predictions += (preds == labels_y).sum()
        no_of_samples += preds.size(0)

    training_loss = (running_loss / dataset_size)

    train_accuracy = (no_of_correct_predictions.item()/no_of_samples)*100

    for name, param in model.named_parameters():
        model_parameters[name] = param

    return training_loss, train_accuracy
```

In the above function I predict the training loss and accuracy by sending my trainloader. I also store the weights from each dataset in a dictionary, where I get the weights for each layers of the architecture.

```
def test(model, criterion, epoch, dataloader, dataset_size):
    valid_loss = 0.0
    no_of_correct_predictions = no_of_samples = 0

    model.eval()

    with torch.no_grad():
        for data in dataloader:
            input_x, labels_y  = data
            batch_size = input_x.shape[0]

            input_x = input_x.to(device)
            labels_y = labels_y.to(device)
            outputs_y_hat = model(input_x)

            loss = criterion(outputs_y_hat, labels_y)

            valid_loss += loss.data * batch_size

            _, preds = outputs_y_hat.max(1)
            no_of_correct_predictions += (preds == labels_y).sum()
            no_of_samples += preds.size(0)


    test_loss = (valid_loss / dataset_size)
    print("Test loss calculated after updating the weights of the model :", test_loss.item())

    test_accuracy = (no_of_correct_predictions.item()/no_of_samples)*100
    print("Test Accuracy calculated after updating the weights of the model :", test_accuracy)
    return
```

In this function I calculate the total test loss and accuracy after updating the model with new weights.

```
def gradients_average():
    for key in model_parameters.keys():
        if model_parameters[key] is not None:
            new_param = comm.reduce(model_parameters[key], op=MPI.SUM, root=0)
            if new_param is not None:
                global_gradients[key] = (new_param/size)

def new_updated_model():
    for name, param in model.named_parameters():
        param.data = global_gradients[name]
```

In the above I run a loop on the values for each layer of the architecture. I check if the values in that particular layer are not none and when this condition satisfies

I take the sum of all the weights obtained from all the ranks and store in in new_param. Next to this step I find out the average or global weights, that will be further used to test the model on testing set, by dividing the values by size or number of processes.

Then I define another function where I update my model with the new found global weights.

After I have defined all the functions now, let's work in parallel:

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

split_data = None
partitioned_dataset = None
training_loss = 0
train_accuracy = 0

model_parameters = global_gradients = {}

train_dataset = datasets.MNIST(root='dataset/', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='dataset/', train=False, transform=transforms.ToTensor(), download=True)

if rank == 0:
    # train_dataset = datasets.Mnist(root='dataset/', train=True, transform=transforms.ToTensor(), download=True)
    # test_dataset = datasets.Mnist(root='dataset/', train=False, transform=transforms.ToTensor(), download=True)
    split_data = random_split(train_dataset, [int(len(train_dataset)/size) for i in range(size)])
else:
    split_data = None
```

I have stored the number of ranks and processes in two variables rank and size. I have initialised a variable split_data which I'll use later to store my split dataset. I have also initialised two dictionaries to store weights for each layer of the architecture. I have downloaded my MNIST dataset and divided it into training and testing. In my master node I split my training dataset into equal parts by dividing the total length of the dataset by the total number of processes.

```
partitioned_dataset = comm.scatter(split_data, root=0)
print(f'rank:{rank} got this size of training Data:{len(partitioned_dataset)}')

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
learning_rate = 0.01
num_epochs = 15

model = Mnist().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_loader = DataLoader(dataset=partitioned_dataset, batch_size=50, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=50, shuffle=True)
```

Then I sent my split datasets to all the ranks by using comm.scatter. I check for my device. I fixed my learning rate to 0.01 and number of epochs to 15. I called my class MNIST and send it to my device. In this case I have used CrossEntropy as my loss function and SGD as my optimizer. Now in the loader I load my partitioned datasets. So, basically every rank gets a separate train loader and my model is trained in every rank using their respective loaders. For the test loader I have just used my original test dataset.

```
start_time = MPI.Wtime()
for epoch in range(0, num_epochs):

    training_loss, train_accuracy = train(model, criterion, optimizer, train_loader, len(partitioned_dataset), training_loss, train_accuracy)
    comm.Barrier()

    total_loss = comm.reduce(training_loss, op=MPI.SUM, root=0)
    if total_loss is not None:
        if epoch != num_epochs:
            print("Training Loss after ", epoch, " epoch is ", (total_loss/size).item())

        # else:
        #     print("The Final Training Loss is ", (total_loss/size).item())

    total_accuracy = comm.reduce(train_accuracy, op=MPI.SUM, root=0)
    if total_accuracy is not None:
        if epoch != num_epochs:
            print("Train Accuracy after ", epoch, " epoch is ", total_accuracy/size)
        # else:
        #     print("The Final Training Accuracy is :", total_accuracy/size)


    gradients_average()
    global_gradients = comm.bcast(global_gradients, root=0)
    # print("The new weights obtained after training the model is ", global_gradients)
    new_updated_model()
```

Now I initialised the start time to record the time for each process. Now in the epoch for loop I call my train function to get the training loss and accuracy for all the ranks. I have then declared the comm.Barrier() function so that I have all my losses and accuracies from all the ranks that have been opened. I have then taken the sum of all the losses and accuracies by using comm.reduce function and print it for every epochs. After each epoch I call my average gradients function which calculates my global gradients from all the ranks and updates my model by calling the new_updated_model function.

```
end_time = MPI.Wtime()
Net_time = end_time - start_time
all_processes_time = comm.gather(Net_time, root=0)

if rank == 0:
    test(model, criterion, epoch, test_loader, len(test_dataset))
    times = np.vstack(all_processes_time)
    time_sum = np.sum(times)
    print('Execution Time for processes is %.3f' % time_sum)
```

Then I calculate the net time and gather all the time from all the processes and store it in the variable all_process_time. In the master node I call the test function to test my model with the new updated weights and print the accuracy and loss. I also print the execution time and record it to later plot the speedup graphs.

RESULTS :

For number of process = 1

```
sreyashisaha@Sreyashis-Air Week10 % mpiexec -n 1 python3 exercise1.py
rank:0 got this size of training Data:60000
Training Loss after  0  epoch is  1.577326774597168
Train Accuracy after  0  epoch is  51.251666666666665
Training Loss after  1  epoch is  0.40128687024116516
Train Accuracy after  1  epoch is  87.64666666666666
Training Loss after  2  epoch is  0.2571122348308563
Train Accuracy after  2  epoch is  92.07166666666666
Training Loss after  3  epoch is  0.20859882235527039
Train Accuracy after  3  epoch is  93.61
Training Loss after  4  epoch is  0.18051433563232422
Train Accuracy after  4  epoch is  94.36333333333333
Training Loss after  5  epoch is  0.16040349006652832
Train Accuracy after  5  epoch is  95.045
Training Loss after  6  epoch is  0.14666426181793213
Train Accuracy after  6  epoch is  95.51333333333332
Training Loss after  7  epoch is  0.13536253571510315
Train Accuracy after  7  epoch is  95.83166666666668
Training Loss after  8  epoch is  0.12502875924110413
Train Accuracy after  8  epoch is  96.19
Training Loss after  9  epoch is  0.11498506367206573
Train Accuracy after  9  epoch is  96.475
Training Loss after  10  epoch is  0.10735165327787399
Train Accuracy after  10  epoch is  96.64666666666668
Training Loss after  11  epoch is  0.1027040109038353
Train Accuracy after  11  epoch is  96.85333333333334
Training Loss after  12  epoch is  0.09723664075136185
Train Accuracy after  12  epoch is  96.93166666666667
Training Loss after  13  epoch is  0.09250004589557648
Train Accuracy after  13  epoch is  97.10333333333332
Training Loss after  14  epoch is  0.08757150173187256
Train Accuracy after  14  epoch is  97.20166666666667
Test loss calculated after updating the weights of the model : 0.08524422347545624
Test Accuracy calculated after updating the weights of the model : 98.53
Execution time for 1 process  464.939962
```

Here we can see that after 15 epochs my the accuracy obtained from the training dataset is 97.20% and the testing accuracy is 98.53%. The total execution time for 1 process is 464.939962 seconds.

For number of process = 2

```
sreyashisaha@Sreyashis-Air Week10 % mpiexec -n 2 python3 exercise
1.py
rank:1 got this size of training Data:30000
rank:0 got this size of training Data:30000
Training Loss after  0  epoch is  2.2214133739471436
Train Accuracy after  0  epoch is  27.551666666666666
Training Loss after  1  epoch is  1.9301416873931885
Train Accuracy after  1  epoch is  43.56
Training Loss after  2  epoch is  0.8188257813453674
Train Accuracy after  2  epoch is  75.06666666666666
Training Loss after  3  epoch is  0.45997750759124756
Train Accuracy after  3  epoch is  85.90166666666667
Training Loss after  4  epoch is  0.3467877507209778
Train Accuracy after  4  epoch is  89.50166666666667
Training Loss after  5  epoch is  0.2908731698989868
Train Accuracy after  5  epoch is  91.07833333333333
Training Loss after  6  epoch is  0.26518312096595764
Train Accuracy after  6  epoch is  91.93166666666667
Training Loss after  7  epoch is  0.24450692534446716
Train Accuracy after  7  epoch is  92.55333333333334
Training Loss after  8  epoch is  0.22049959003925323
Train Accuracy after  8  epoch is  93.25166666666667
Training Loss after  9  epoch is  0.20650622248649597
Train Accuracy after  9  epoch is  93.64666666666666
Training Loss after  10  epoch is  0.1971476674079895
Train Accuracy after  10  epoch is  93.865
Training Loss after  11  epoch is  0.18417510390281677
Train Accuracy after  11  epoch is  94.32333333333332
Training Loss after  12  epoch is  0.17988324165344238
Train Accuracy after  12  epoch is  94.39
Training Loss after  13  epoch is  0.164984792470932
Train Accuracy after  13  epoch is  94.88666666666666
Training Loss after  14  epoch is  0.16294294595718384
Train Accuracy after  14  epoch is  94.99166666666667
Test loss calculated after updating the weights of the model : 0.
17131541669368744
Test Accuracy calculated after updating the weights of the model
: 96.54
Execution time for 2 process  187.142124
```

Here we can see that after 15 epochs my the accuracy obtained from the training dataset is 94.99% and the testing accuracy is 96.54%. The total execution time for 1 process is 187.142124 seconds. This shows that execution is faster when work is done in parallel.

For number of process = 3

```
sreyashisaha@Sreyashis-Air Week10 % mpiexec -n 3 python3 exercise1.py
rank:1 got this size of training Data:20000
rank:2 got this size of training Data:20000
rank:0 got this size of training Data:20000
Training Loss after  0  epoch is  2.257352828979492
Train Accuracy after  0  epoch is  22.060000000000002
Training Loss after  1  epoch is  2.286830186843872
Train Accuracy after  1  epoch is  19.486666666666668
Training Loss after  2  epoch is  2.2183496952056885
Train Accuracy after  2  epoch is  31.145
Training Loss after  3  epoch is  1.7604389190673828
Train Accuracy after  3  epoch is  46.18666666666667
Training Loss after  4  epoch is  1.0726414918899536
Train Accuracy after  4  epoch is  64.34833333333334
Training Loss after  5  epoch is  0.7090215682983398
Train Accuracy after  5  epoch is  77.415
Training Loss after  6  epoch is  0.48988857865333557
Train Accuracy after  6  epoch is  84.87333333333333
Training Loss after  7  epoch is  0.39753270149230957
Train Accuracy after  7  epoch is  87.92666666666666
Training Loss after  8  epoch is  0.3418768346309662
Train Accuracy after  8  epoch is  89.53666666666668
Training Loss after  9  epoch is  0.30874884128570557
Train Accuracy after  9  epoch is  90.605
Training Loss after  10  epoch is  0.28750523924827576
Train Accuracy after  10  epoch is  91.27833333333332
Training Loss after  11  epoch is  0.26192668080329895
Train Accuracy after  11  epoch is  92.06
Training Loss after  12  epoch is  0.24470455944538116
Train Accuracy after  12  epoch is  92.565
Training Loss after  13  epoch is  0.2291955202817917
Train Accuracy after  13  epoch is  93.085
Training Loss after  14  epoch is  0.21700507402420044
Train Accuracy after  14  epoch is  93.34833333333331
Test loss calculated after updating the weights of the model : 0.22378872334957123
Test Accuracy calculated after updating the weights of the model : 95.86
Execution time for 3 process  170.388669
```

Here we can see that after 15 epochs my the accuracy obtained from the training dataset is 93.34% and the testing accuracy is 95.86%. The total execution time for 3 process is 170.388669 seconds. This shows that execution is faster when work is done in parallel.

Similarly for processes 4 and 5 we get the following output:

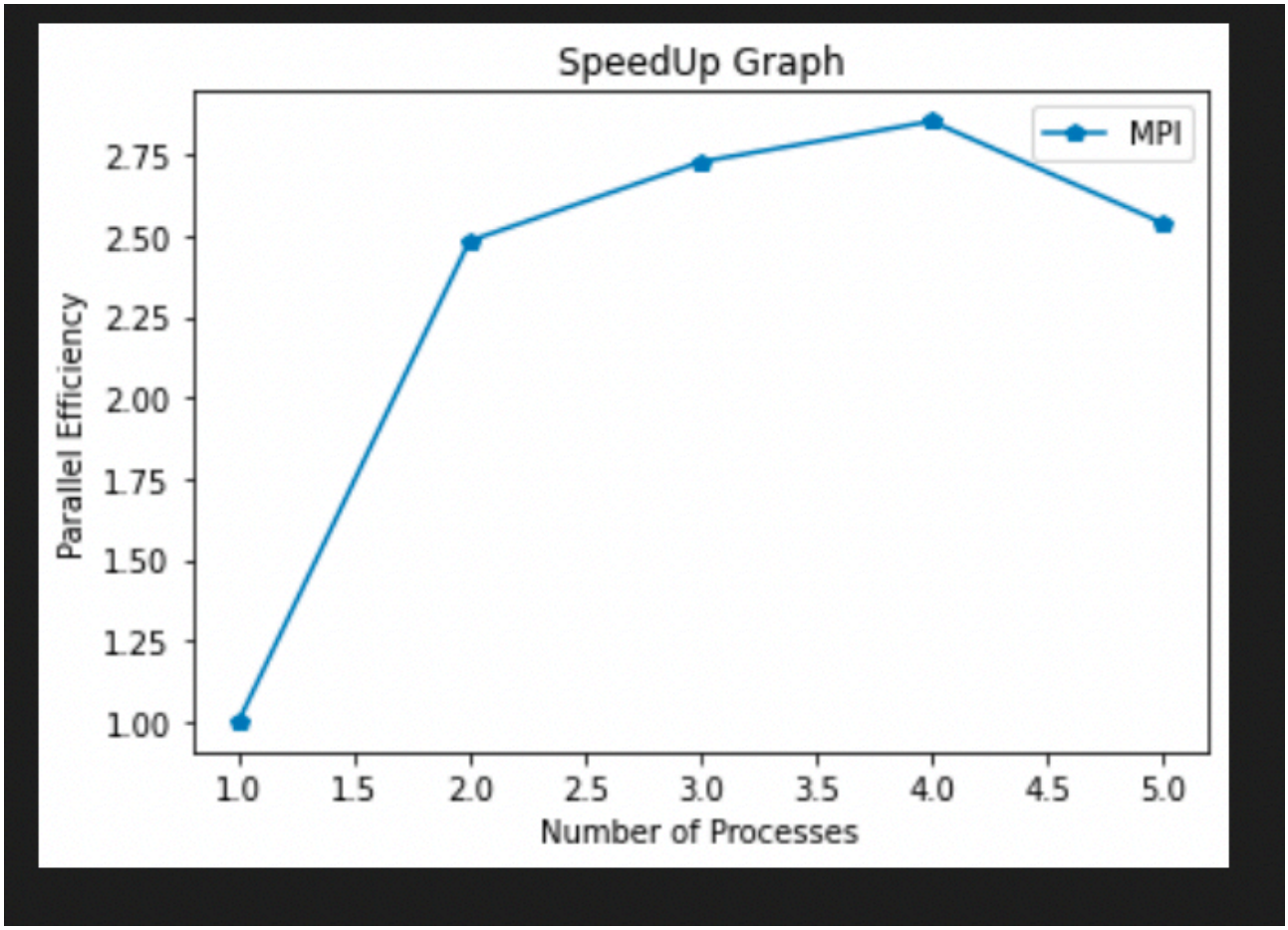
```
sreyashisaha@Sreyashis-Air Week10 % mpiexec -n 4 python3 exercise1.py
rank:1 got this size of training Data:15000
rank:2 got this size of training Data:15000
rank:0 got this size of training Data:15000
rank:3 got this size of training Data:15000
Training Loss after  0  epoch is  2.2820982933044434
Train Accuracy after  0  epoch is  17.453333333333333
Training Loss after  1  epoch is  2.301251173019409
Train Accuracy after  1  epoch is  10.238333333333333
Training Loss after  2  epoch is  2.299565076828003
Train Accuracy after  2  epoch is  13.28
Training Loss after  3  epoch is  2.2972779273986816
Train Accuracy after  3  epoch is  11.351666666666668
Training Loss after  4  epoch is  2.2929513454437256
Train Accuracy after  4  epoch is  15.156666666666666
Training Loss after  5  epoch is  2.282416343688965
Train Accuracy after  5  epoch is  20.056666666666665
Training Loss after  6  epoch is  2.2451910972595215
Train Accuracy after  6  epoch is  22.15
Training Loss after  7  epoch is  2.040731906890869
Train Accuracy after  7  epoch is  26.27166666666667
Training Loss after  8  epoch is  1.7375881671905518
Train Accuracy after  8  epoch is  36.75833333333334
Training Loss after  9  epoch is  1.5021934509277344
Train Accuracy after  9  epoch is  49.351666666666674
Training Loss after  10  epoch is  1.0366733074188232
Train Accuracy after  10  epoch is  65.41833333333332
Training Loss after  11  epoch is  0.7471692562103271
Train Accuracy after  11  epoch is  74.91333333333334
Training Loss after  12  epoch is  0.5805416107177734
Train Accuracy after  12  epoch is  81.41999999999999
Training Loss after  13  epoch is  0.47759127616882324
Train Accuracy after  13  epoch is  84.985
Training Loss after  14  epoch is  0.40387314558029175
Train Accuracy after  14  epoch is  87.48166666666667
Test loss calculated after updating the weights of the model : 0.40371352434158325
Test Accuracy calculated after updating the weights of the model : 93.0
Execution time for 4 process  162.914951
```

```
sreyashisaha@Sreyashis-Air Week10 % mpiexec -n 5 python3 exercise1.py
rank:1 got this size of training Data:12000
rank:2 got this size of training Data:12000
rank:3 got this size of training Data:12000
rank:0 got this size of training Data:12000
rank:4 got this size of training Data:12000
Training Loss after  0  epoch is  2.286017894744873
Train Accuracy after  0  epoch is  14.804999999999998
Training Loss after  1  epoch is  2.301591396331787
Train Accuracy after  1  epoch is  11.576666666666668
Training Loss after  2  epoch is  2.3003525733947754
Train Accuracy after  2  epoch is  11.241666666666669
Training Loss after  3  epoch is  2.2991175651550293
Train Accuracy after  3  epoch is  11.236666666666668
Training Loss after  4  epoch is  2.297569751739502
Train Accuracy after  4  epoch is  11.236666666666668
Training Loss after  5  epoch is  2.2952871322631836
Train Accuracy after  5  epoch is  11.355
Training Loss after  6  epoch is  2.2912821769714355
Train Accuracy after  6  epoch is  14.615
Training Loss after  7  epoch is  2.2831714153289795
Train Accuracy after  7  epoch is  21.816666666666666
Training Loss after  8  epoch is  2.263561248779297
Train Accuracy after  8  epoch is  22.315
Training Loss after  9  epoch is  2.1939806938171387
Train Accuracy after  9  epoch is  25.169999999999998
Training Loss after  10  epoch is  1.9837726354599
Train Accuracy after  10  epoch is  32.085
Training Loss after  11  epoch is  1.7523252964019775
Train Accuracy after  11  epoch is  41.75166666666667
Training Loss after  12  epoch is  1.3652764558792114
Train Accuracy after  12  epoch is  56.48166666666667
Training Loss after  13  epoch is  1.0035592317581177
Train Accuracy after  13  epoch is  67.29833333333333
Training Loss after  14  epoch is  0.7940323948860168
Train Accuracy after  14  epoch is  73.87333333333333
Test loss calculated after updating the weights of the model : 0.7659114003181458
Test Accuracy calculated after updating the weights of the model : 86.42999999999999
Execution time for  0  process is  183.040793
```

Now after calculating the time I used those to plot the speedup graph.

```python
from matplotlib import pyplot as plt

processes=[1,2,3,4,5]
plt.plot(processes, [464.939962/464.939962, 464.939962/187.142124, 464.939962/170.388669,  464.939962/162.914951, 464.939962/183.040793], label='MPI', marker="p")
plt.title('SpeedUp Graph')
plt.xlabel('Number of Processes')
plt.ylabel('Parallel Efficiency')
plt.legend()
plt.show()
```



EXERCISE 2:

```
from torchvision import datasets, transforms
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import CrossEntropyLoss
from torch.optim import SGD
import torch.multiprocessing as mp
from torch.utils.data import DataLoader, DistributedSampler
import time
from zmq import device
```

Importing all the libraries to implement HogWild SGD along with multiprocessing.

I have made slight changes in the model and implemented it as asked in the question.

```
class Mnist(nn.Module):
  def __init__(self):
    super(Mnist, self).__init__()

    self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=5, stride=1, padding = 1)
    self.pool1 = nn.MaxPool2d(kernel_size=2, stride = 2, padding = 1)
    self.conv2 = nn.Conv2d(in_channels=16, out_channels=64, kernel_size=5, stride=1, padding = 0)
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride = 2)
    # self.pool3 = nn.MaxPool2d(kernel_size=5, stride = 1)
    # self.fc1 = nn.Linear(in_features=64, out_features=32)
    self.fc2 = nn.Linear(in_features=64, out_features=10)
    # self.dp1 = nn.Dropout(0.5)

  def forward(self, x):
    x = F.relu(self.conv1(x))
    x = self.pool1(x)
    x = F.relu(self.conv2(x))
    x = self.pool2(x)
    # x = self.dp1(x)
    # x = self.pool3(x)
    x = x.reshape(x.shape[0], -1)
    # x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x
```

Here I have removed 1 maxpool, dropout and a fully connected layer as it is not required when we use multiprocessing with pytorch.

I have defined certain functions to calculate the accuracy of the test and train models. In the function below I calculate the accuracy of the input dataset and return it. Here X is the tensors and y_true are basically the labels of those tensors.

So.

```
def get_accuracy(model, data_loader, device):
    correct_pred = 0
    n = 0

    with torch.no_grad():
        for X, y_true in data_loader:
            X = X.to(device)
            y_true = y_true.to(device)

            y_prob = model(X)
            _, predicted_labels = torch.max(y_prob, 1)

            n += y_true.size(0)
            correct_pred += (predicted_labels == y_true).sum()

    return correct_pred.float() / n
```

Basically I check whether my predicted labels match the original label or not and find the accuracy.

```
def test_model(model, data_loader, device, criterion):
    model.eval()
    running_loss = 0

    for X, y_true in data_loader:
        X = X.to(device)
        y_true = y_true.to(device)

        # Forward pass and record loss
        y_hat = model(X)
        loss = criterion(y_hat, y_true)
        running_loss += loss.item() * X.size(0)

    Test_Loss = running_loss / len(data_loader.dataset)
    Test_acc = get_accuracy(model, data_loader, device)

    print(f'Test Loss: {Test_Loss},  Test Accuracy: {Test_acc} ')
```

In the above function I calculate my loss and accuracy for the test dataset.

Moreover, I have defined a training function, which is taking a model and a data loader as input parameters. Inside of this function, the SGD optimizer is used and the already mentioned CrossEntropyLoss. This training function is a standard implementation of a PyTorch program. Each process participating in Hogwild will call it at the same time.

```
def train(model, data_loader, rank, opt,device):
    # defining the optimizer
    # defining loss function
    loss_fn = CrossEntropyLoss()
    epochs = 15
    for epoch in range(epochs):
        model.train()
        for idx, (train_x, train_label) in enumerate(data_loader):
            opt.zero_grad()
            predict_y = model(train_x.float())
            loss = loss_fn(predict_y, train_label.long())
            loss.backward()
            opt.step()
        Train_acc = get_accuracy(model, data_loader, device) * 100
        print(f'Epoch:{epoch} Training Loss: {loss}, Training Accuracy: {Train_acc} in rank:{rank}')
```

So I'll get the loss and accuracy values for each of the process.

```
if __name__ == '__main__':
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    criterion = torch.nn.CrossEntropyLoss()
    model = Mnist()
    optimizer = SGD(model.parameters(), lr=0.01, momentum=0.9)
    start_time = time.time()
    num_processes = 5
    model.share_memory()
    transform = transforms.ToTensor()
    train_dataset = datasets.MNIST('./data/MNIST', train=True, download=True,
                                   transform=transform)
    test_dataset = datasets.MNIST('./data/MNIST', train=False,
                                  transform=transform)

    processes = []
    for rank in range(num_processes):
        train_loader = DataLoader(dataset=train_dataset,
                                  sampler=DistributedSampler(dataset=train_dataset, num_replicas=num_processes,
                                                             rank=rank), batch_size=256)
        p = mp.Process(target=train, args=(model, train_loader, rank, optimizer,device))
        p.start()
        processes.append(p)
    for p in processes:
        p.join()
```

The above code part contains the parallel setup. In this case, we use the single method call share memory to instantiate the model, provide the number of parallel processes, and push it into shared memory. The Mint dataset, which is included in the torchvision package, is the dataset that was used.
We define a data loader for each process after looping over the processes. The data loaders contain a distributed sampler that manages the distribution of the data and is aware of the process rank. Each process thus has its own data partition. In each process, the multiprocessing package invokes the training function, and then uses the join command to wait for the processes to complete. All processes have access to the shared model when training, but they only use their own data partition. As a result, we reduce training time by about 4, which is the total number of training processes.

```
test_model(model, DataLoader(dataset=test_dataset, batch_size=256), device, criterion)
end_time = time.time()
Execution_time = end_time - start_time
print(f'Total Execution Time: {Execution_time}')
```

After training the model I test it by calling the test function and note the final test loss and accuracy. I also calculate the time taken as I increase my processes from 1 to 5. I then use it to plot the speedup graph.

RESULTS:

For process = 1

```
Epoch:0 Training Loss: 0.5414305329322815, Training Accuracy: 87.1050033569336 in rank:0
Epoch:1 Training Loss: 0.28843459486961365, Training Accuracy: 93.82499694824219 in rank:0
Epoch:2 Training Loss: 0.25985994935035706, Training Accuracy: 93.44999694824219 in rank:0
Epoch:3 Training Loss: 0.3028411865234375, Training Accuracy: 95.25499725341797 in rank:0
Epoch:4 Training Loss: 0.19590318202972412, Training Accuracy: 96.07666778564453 in rank:0
Epoch:5 Training Loss: 0.12681922316551208, Training Accuracy: 96.45166015625 in rank:0
Epoch:6 Training Loss: 0.17020989954471588, Training Accuracy: 96.90666961669922 in rank:0
Epoch:7 Training Loss: 0.13411958515644073, Training Accuracy: 97.11000061035156 in rank:0
Epoch:8 Training Loss: 0.1016564667224884, Training Accuracy: 97.39999389648438 in rank:0
Epoch:9 Training Loss: 0.08883198350667953, Training Accuracy: 97.38500213623047 in rank:0
Epoch:10 Training Loss: 0.05875344201922417, Training Accuracy: 97.63500213623047 in rank:0
Epoch:11 Training Loss: 0.1204620897769928, Training Accuracy: 97.4749984741211 in rank:0
Epoch:12 Training Loss: 0.10601354390382767, Training Accuracy: 97.43000030517578 in rank:0
Epoch:13 Training Loss: 0.06990814954042435, Training Accuracy: 97.77166748046875 in rank:0
Epoch:14 Training Loss: 0.07577936351299286, Training Accuracy: 97.89500427246094 in rank:0
Test Loss: 0.07385110346525907,  Test Accuracy: 0.9850000143051147
Total Execution Time: 303.7631859779358
```

Here we observe the train loss and accuracy for just 1 process and record the time which is 303.7632 seconds approximately.

For process = 2

```
sreyashisaha@Sreyashis-Air Week10 % python3 exercise2.py
Epoch:0 Training Loss: 1.29646897315979, Training Accuracy: 51.293331146240234 in rank:1
Epoch:0 Training Loss: 1.619484782218933, Training Accuracy: 50.98999786376953 in rank:0
Epoch:1 Training Loss: 0.1839684247970581, Training Accuracy: 89.96666717529297 in rank:1
Epoch:1 Training Loss: 0.374401718378067, Training Accuracy: 90.00666809082031 in rank:0
Epoch:2 Training Loss: 0.16386710107326508, Training Accuracy: 93.66000366210938 in rank:1
Epoch:2 Training Loss: 0.28742119669914246, Training Accuracy: 93.77666473388672 in rank:0
Epoch:3 Training Loss: 0.09850289672613144, Training Accuracy: 94.663330078125 in rank:1
Epoch:3 Training Loss: 0.23977775871753693, Training Accuracy: 94.69666290283203 in rank:0
Epoch:4 Training Loss: 0.16315880417823792, Training Accuracy: 95.4800033569336 in rank:1
Epoch:4 Training Loss: 0.2423732876777649, Training Accuracy: 95.51666259765625 in rank:0
Epoch:5 Training Loss: 0.06010204181075096, Training Accuracy: 96.11333465576172 in rank:1
Epoch:5 Training Loss: 0.32647886872291565, Training Accuracy: 96.29000091552734 in rank:0
Epoch:6 Training Loss: 0.0767005905508995, Training Accuracy: 96.34333038330078 in rank:1
Epoch:6 Training Loss: 0.12031661719083786, Training Accuracy: 96.5999984741211 in rank:0
Epoch:7 Training Loss: 0.07261934131383896, Training Accuracy: 96.86000061035156 in rank:1
Epoch:7 Training Loss: 0.11489135771989822, Training Accuracy: 96.94999694824219 in rank:0
Epoch:8 Training Loss: 0.07797636836767197, Training Accuracy: 96.94667053222656 in rank:1
Epoch:8 Training Loss: 0.15159951150417328, Training Accuracy: 96.92666625976562 in rank:0
Epoch:9 Training Loss: 0.05523093417286873, Training Accuracy: 97.15666961669922 in rank:1
Epoch:9 Training Loss: 0.1410164088010788, Training Accuracy: 97.37667083740234 in rank:0
Epoch:10 Training Loss: 0.196416974067688, Training Accuracy: 97.49333190917969 in rank:0
Epoch:10 Training Loss: 0.09458162635564804, Training Accuracy: 97.47666931152344 in rank:1
Epoch:11 Training Loss: 0.02625282295048237, Training Accuracy: 97.44667053222656 in rank:0
Epoch:11 Training Loss: 0.05733175575733185, Training Accuracy: 97.51667022705078 in rank:1
Epoch:12 Training Loss: 0.1265455037355423, Training Accuracy: 97.61666870117188 in rank:0
Epoch:12 Training Loss: 0.03224969282746315, Training Accuracy: 97.77667236328125 in rank:1
Epoch:13 Training Loss: 0.03634091094136238, Training Accuracy: 97.29000091552734 in rank:0
Epoch:13 Training Loss: 0.08094926923513412, Training Accuracy: 97.47000122070312 in rank:1
Epoch:14 Training Loss: 0.1236233189702034, Training Accuracy: 97.59667205810547 in rank:0
Epoch:14 Training Loss: 0.03247431293129921, Training Accuracy: 97.77667236328125 in rank:1
Test Loss: 0.07461836634278297,  Test Accuracy: 0.986299991607666
Total Execution Time: 257.5363759994507
```

Here we observe the loss and accuracies for the training dataset for 2 processes that is for rank 0 and rank 1. And in addition to this we also record the total execution time which is decreasing as we increase the number of processes.

Similarly for process = 3,4,5 we get the following results:

```
sreyashisaha@Sreyashis-Air Week10 % python3 exercise2.py
Epoch:0 Training Loss: 1.1071772575378418, Training Accuracy: 79.36499786376953 in rank:0
Epoch:0 Training Loss: 0.653856635093689, Training Accuracy: 79.3499984741211 in rank:1
Epoch:0 Training Loss: 1.1847728490829468, Training Accuracy: 79.39500427246094 in rank:2
Epoch:1 Training Loss: 0.2243276834487915, Training Accuracy: 92.74500274658203 in rank:0
Epoch:1 Training Loss: 0.14437177777290344, Training Accuracy: 93.01499938964844 in rank:1
Epoch:1 Training Loss: 0.19228824973106384, Training Accuracy: 92.94000244140625 in rank:2
Epoch:2 Training Loss: 0.2317628264427185, Training Accuracy: 94.875 in rank:0
Epoch:2 Training Loss: 0.060614682734012604, Training Accuracy: 94.73999786376953 in rank:1
Epoch:2 Training Loss: 0.1330387443304062, Training Accuracy: 95.01499938964844 in rank:2
Epoch:3 Training Loss: 0.05087300017476082, Training Accuracy: 95.81500244140625 in rank:0
Epoch:3 Training Loss: 0.10073842853307724, Training Accuracy: 95.76000213623047 in rank:1
Epoch:3 Training Loss: 0.12344850599765778, Training Accuracy: 95.54499816894531 in rank:2
Epoch:4 Training Loss: 0.05117189139127731, Training Accuracy: 96.57499694824219 in rank:1
Epoch:4 Training Loss: 0.12886570394039154, Training Accuracy: 96.55000305175781 in rank:0
Epoch:4 Training Loss: 0.1294594407081604, Training Accuracy: 96.67500305175781 in rank:2
Epoch:5 Training Loss: 0.13665546476840973, Training Accuracy: 96.59500122070312 in rank:0
Epoch:5 Training Loss: 0.0940496101975441, Training Accuracy: 96.45999908447266 in rank:1
Epoch:5 Training Loss: 0.12077143043279648, Training Accuracy: 96.70500183105469 in rank:2
Epoch:6 Training Loss: 0.03856400400400162, Training Accuracy: 97.15999603271484 in rank:0
Epoch:6 Training Loss: 0.0767010971903801, Training Accuracy: 97.15999603271484 in rank:1
Epoch:6 Training Loss: 0.04313793405890465, Training Accuracy: 97.18499755859375 in rank:2
Epoch:7 Training Loss: 0.01804659515619278, Training Accuracy: 97.18000030517578 in rank:0
Epoch:7 Training Loss: 0.043727174401283264, Training Accuracy: 97.01000213623047 in rank:1
Epoch:7 Training Loss: 0.08519534021615982, Training Accuracy: 97.18000030517578 in rank:2
Epoch:8 Training Loss: 0.03777868300676346, Training Accuracy: 97.65499877929688 in rank:0
Epoch:8 Training Loss: 0.052749864757061005, Training Accuracy: 97.62999725341797 in rank:1
Epoch:8 Training Loss: 0.042665909975767136, Training Accuracy: 97.64999389648438 in rank:2
Epoch:9 Training Loss: 0.10716764628887177, Training Accuracy: 97.5199966430664 in rank:0
Epoch:9 Training Loss: 0.024779105558991432, Training Accuracy: 97.69999694824219 in rank:1
Epoch:9 Training Loss: 0.044897209852933884, Training Accuracy: 97.68000030517578 in rank:2
Epoch:10 Training Loss: 0.036971352994441986, Training Accuracy: 97.76499938964844 in rank:0
Epoch:10 Training Loss: 0.020547829568386078, Training Accuracy: 97.77999877929688 in rank:1
Epoch:10 Training Loss: 0.10726343095302582, Training Accuracy: 98.0 in rank:2
Epoch:11 Training Loss: 0.059892598539590836, Training Accuracy: 97.81999969482422 in rank:0
Epoch:11 Training Loss: 0.025197375565767288, Training Accuracy: 97.90999603271484 in rank:1
Epoch:11 Training Loss: 0.07896065711975098, Training Accuracy: 97.86000061035156 in rank:2
Epoch:12 Training Loss: 0.10391256958246231, Training Accuracy: 97.82500457763672 in rank:0
Epoch:12 Training Loss: 0.010657843202352524, Training Accuracy: 97.94499969482422 in rank:1
Epoch:12 Training Loss: 0.06787731498479843, Training Accuracy: 97.91999816894531 in rank:2
Epoch:13 Training Loss: 0.017451664432883263, Training Accuracy: 97.94499969482422 in rank:0
Epoch:13 Training Loss: 0.03771093115210533, Training Accuracy: 98.03500366210938 in rank:1
Epoch:13 Training Loss: 0.03034125454723835, Training Accuracy: 98.0999984741211 in rank:2
Epoch:14 Training Loss: 0.08304228633642197, Training Accuracy: 97.93499755859375 in rank:0
Epoch:14 Training Loss: 0.05247792229056358, Training Accuracy: 97.97000122070312 in rank:1
Epoch:14 Training Loss: 0.04882875084877014, Training Accuracy: 98.04499816894531 in rank:2
Test Loss: 0.06565631391555071,  Test Accuracy: 0.9868000149726868
Total Execution Time: 236.5158190727234
```

We can observe that overall the execution time decreases, however, we do observe an increase in the execution time as I increase the number of process to 4.

```
sreyashisaha@Sreyashis-Air Week10 % python3 exercise2.py
Epoch:0 Training Loss: 2.2500267028808594, Training Accuracy: 20.613332748413086 in rank:3
Epoch:0 Training Loss: 2.244638681411743, Training Accuracy: 20.739999771118164 in rank:0
Epoch:0 Training Loss: 2.2226641178131104, Training Accuracy: 20.559999465942383 in rank:1
Epoch:0 Training Loss: 2.22866153717041, Training Accuracy: 20.3266658782959 in rank:2
Epoch:1 Training Loss: 1.3700915575027466, Training Accuracy: 54.36666488647461 in rank:0
Epoch:1 Training Loss: 1.2291812896728516, Training Accuracy: 56.00666427612305 in rank:3
Epoch:1 Training Loss: 1.432437777519226, Training Accuracy: 55.013336181640625 in rank:1
Epoch:1 Training Loss: 1.402391791343689, Training Accuracy: 55.193336486816406 in rank:2
Epoch:2 Training Loss: 0.4675127863883972, Training Accuracy: 84.24666595458984 in rank:0
Epoch:2 Training Loss: 0.4334412217140198, Training Accuracy: 84.30000305175781 in rank:3
Epoch:2 Training Loss: 0.5851912498474121, Training Accuracy: 84.11333465576172 in rank:1
Epoch:2 Training Loss: 0.7321030497550964, Training Accuracy: 84.82666778564453 in rank:2
Epoch:3 Training Loss: 0.2109355926513672, Training Accuracy: 90.73999786376953 in rank:0
Epoch:3 Training Loss: 0.21315710246562958, Training Accuracy: 91.19999694824219 in rank:3
Epoch:3 Training Loss: 0.3727662265300751, Training Accuracy: 90.95999908447266 in rank:1
Epoch:3 Training Loss: 0.33673396706581116, Training Accuracy: 91.32666778564453 in rank:2
Epoch:4 Training Loss: 0.11456327885389328, Training Accuracy: 93.69999694824219 in rank:0
Epoch:4 Training Loss: 0.2819373607635498, Training Accuracy: 93.36000061035156 in rank:1
Epoch:4 Training Loss: 0.20415471494197845, Training Accuracy: 94.1933364868164 in rank:2
Epoch:4 Training Loss: 0.20059648156166077, Training Accuracy: 93.87999725341797 in rank:3
Epoch:5 Training Loss: 0.2880505621433258, Training Accuracy: 94.37999725341797 in rank:1
Epoch:5 Training Loss: 0.15195414423942566, Training Accuracy: 94.57333374023438 in rank:0
Epoch:5 Training Loss: 0.2346731722354889, Training Accuracy: 94.44666290283203 in rank:2
Epoch:5 Training Loss: 0.13900023698806763, Training Accuracy: 94.66666412353516 in rank:3
Epoch:6 Training Loss: 0.2742518484592438, Training Accuracy: 95.07333374023438 in rank:1
Epoch:6 Training Loss: 0.09194972366094589, Training Accuracy: 95.17333221435547 in rank:0
Epoch:6 Training Loss: 0.2643096446990967, Training Accuracy: 95.25333404541016 in rank:2
Epoch:6 Training Loss: 0.14056558907032013, Training Accuracy: 95.00666809082031 in rank:3
Epoch:7 Training Loss: 0.16170388460159302, Training Accuracy: 95.45999908447266 in rank:1
Epoch:7 Training Loss: 0.19789017736911774, Training Accuracy: 95.76667022705078 in rank:2
Epoch:7 Training Loss: 0.0823778510093689, Training Accuracy: 95.81999969482422 in rank:0
Epoch:7 Training Loss: 0.14972828328609467, Training Accuracy: 95.69332885742188 in rank:3
Epoch:8 Training Loss: 0.18290887773036957, Training Accuracy: 95.5133285522461 in rank:1
Epoch:8 Training Loss: 0.12404347956180573, Training Accuracy: 95.42000579833984 in rank:0
Epoch:8 Training Loss: 0.13850097358226776, Training Accuracy: 95.586669921875 in rank:3
Epoch:8 Training Loss: 0.2243398129940033, Training Accuracy: 95.72666931152344 in rank:2
Epoch:9 Training Loss: 0.1582823097705841, Training Accuracy: 95.94000244140625 in rank:1
Epoch:9 Training Loss: 0.061655547469854355, Training Accuracy: 96.20000457763672 in rank:0
Epoch:9 Training Loss: 0.14073503017425537, Training Accuracy: 96.34666442871094 in rank:3
Epoch:9 Training Loss: 0.20951330661773682, Training Accuracy: 96.26000213623047 in rank:2
Epoch:10 Training Loss: 0.13913536071777344, Training Accuracy: 96.02666473388672 in rank:1
Epoch:10 Training Loss: 0.08288881927728653, Training Accuracy: 96.24666595458984 in rank:0
Epoch:10 Training Loss: 0.15149155259132385, Training Accuracy: 96.09333038330078 in rank:3
Epoch:10 Training Loss: 0.18129687011241913, Training Accuracy: 96.24666595458984 in rank:2
Epoch:11 Training Loss: 0.13250403106212616, Training Accuracy: 96.41999816894531 in rank:1
Epoch:11 Training Loss: 0.06339661777019501, Training Accuracy: 96.63333129882812 in rank:0
Epoch:11 Training Loss: 0.11818728595972061, Training Accuracy: 96.43333435058594 in rank:3
Epoch:11 Training Loss: 0.11887563019990921, Training Accuracy: 96.59333038330078 in rank:2
Epoch:12 Training Loss: 0.12128116190433502, Training Accuracy: 95.94000244140625 in rank:1
Epoch:12 Training Loss: 0.10452735424041748, Training Accuracy: 96.37332916259766 in rank:0
Epoch:12 Training Loss: 0.10924117267131805, Training Accuracy: 96.21333312988281 in rank:3
Epoch:12 Training Loss: 0.18974675238132477, Training Accuracy: 96.38666534423828 in rank:2
Epoch:13 Training Loss: 0.10761354863643646, Training Accuracy: 96.88666534423828 in rank:1
Epoch:13 Training Loss: 0.08790196478366852, Training Accuracy: 96.83999633789062 in rank:0
Epoch:13 Training Loss: 0.09786517173051834, Training Accuracy: 97.05332946777344 in rank:3
Epoch:13 Training Loss: 0.13129596412181854, Training Accuracy: 97.18000030517578 in rank:2
Epoch:14 Training Loss: 0.14660990238189697, Training Accuracy: 96.68000030517578 in rank:1
Epoch:14 Training Loss: 0.07015048712491989, Training Accuracy: 96.73999786376953 in rank:0
Epoch:14 Training Loss: 0.04693705961108208, Training Accuracy: 97.1199951171875 in rank:3
Epoch:14 Training Loss: 0.09554847329854965, Training Accuracy: 96.8933334350586 in rank:2
Test Loss: 0.1069088090211153,  Test Accuracy: 0.977400004863739
Total Execution Time: 239.90274691581726
```

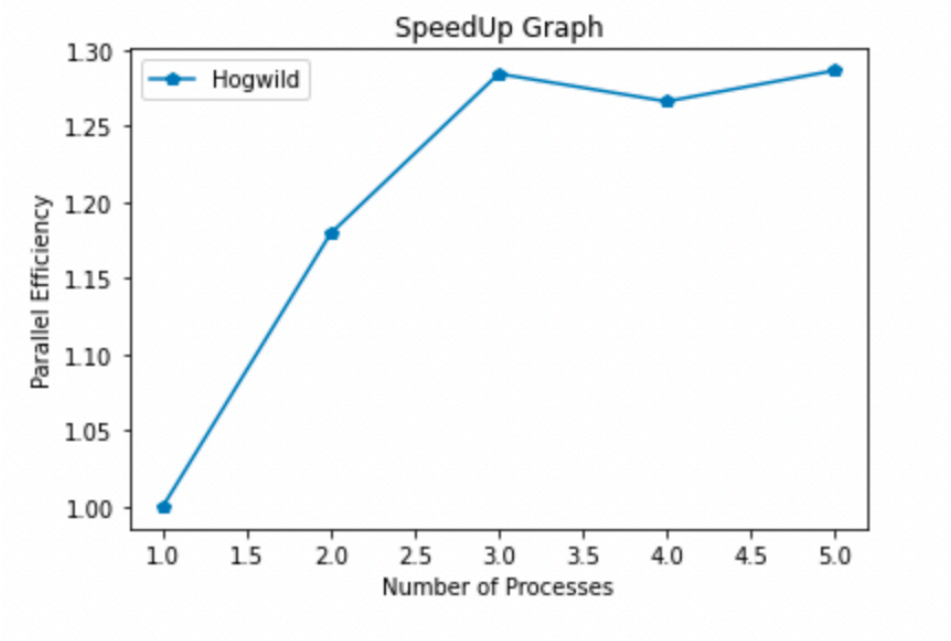
Here the time increased from 236 seconds in 3 processes to 239 seconds for 4 processes.

```
Epoch:7 Training Loss: 0.6724348664283752, Training Accuracy: 79.4749984741211 in rank:0
Epoch:7 Training Loss: 0.5777102112770081, Training Accuracy: 79.99166870117188 in rank:1
Epoch:7 Training Loss: 0.6777562499046326, Training Accuracy: 79.67500305175781 in rank:3
Epoch:7 Training Loss: 0.6238378882408142, Training Accuracy: 80.26666259765625 in rank:2
Epoch:7 Training Loss: 0.6637340188026428, Training Accuracy: 79.59166717529297 in rank:4
Epoch:8 Training Loss: 0.5411608815193176, Training Accuracy: 81.74166870117188 in rank:0
Epoch:8 Training Loss: 0.5700650811195374, Training Accuracy: 81.94166564941406 in rank:3
Epoch:8 Training Loss: 0.5867621302604675, Training Accuracy: 82.49166870117188 in rank:1
Epoch:8 Training Loss: 0.49247393012046814, Training Accuracy: 82.7750015258789 in rank:2
Epoch:8 Training Loss: 0.6141222715377808, Training Accuracy: 81.91666412353516 in rank:4
Epoch:9 Training Loss: 0.5904189944267273, Training Accuracy: 82.83332824707031 in rank:0
Epoch:9 Training Loss: 0.6328058838844299, Training Accuracy: 83.30000305175781 in rank:3
Epoch:9 Training Loss: 0.5835152268409729, Training Accuracy: 83.125 in rank:1
Epoch:9 Training Loss: 0.5344353318214417, Training Accuracy: 83.10833740234375 in rank:2
Epoch:9 Training Loss: 0.5158776044845581, Training Accuracy: 83.10833740234375 in rank:4
Epoch:10 Training Loss: 0.5924047827720642, Training Accuracy: 83.40833282470703 in rank:0
Epoch:10 Training Loss: 0.44246339797973633, Training Accuracy: 83.84166717529297 in rank:1
Epoch:10 Training Loss: 0.5880461931228638, Training Accuracy: 84.19999694824219 in rank:3
Epoch:10 Training Loss: 0.37864193320274353, Training Accuracy: 84.3499984741211 in rank:2
Epoch:10 Training Loss: 0.4832060933113098, Training Accuracy: 84.3499984741211 in rank:4
Epoch:11 Training Loss: 0.451078325510025, Training Accuracy: 84.65833282470703 in rank:0
Epoch:11 Training Loss: 0.5824899077415466, Training Accuracy: 84.67500305175781 in rank:3
Epoch:11 Training Loss: 0.4123404324054718, Training Accuracy: 85.41667175292969 in rank:1
Epoch:11 Training Loss: 0.37250080704689026, Training Accuracy: 85.40833282470703 in rank:2
Epoch:11 Training Loss: 0.49608114361763, Training Accuracy: 84.93333435058594 in rank:4
Epoch:12 Training Loss: 0.38881805539131165, Training Accuracy: 83.625 in rank:0
Epoch:12 Training Loss: 0.45855727791786194, Training Accuracy: 83.67499542236328 in rank:1
Epoch:12 Training Loss: 0.5371359586715698, Training Accuracy: 84.20833587646484 in rank:3
Epoch:12 Training Loss: 0.3554553687572479, Training Accuracy: 84.54166412353516 in rank:2
Epoch:12 Training Loss: 0.554020881652832, Training Accuracy: 83.75 in rank:4
Epoch:13 Training Loss: 0.4398791790008545, Training Accuracy: 86.56666564941406 in rank:0
Epoch:13 Training Loss: 0.4260326027870178, Training Accuracy: 86.69999694824219 in rank:1
Epoch:13 Training Loss: 0.5166801810264587, Training Accuracy: 87.49166870117188 in rank:3
Epoch:13 Training Loss: 0.4671877324581146, Training Accuracy: 87.46666717529297 in rank:2
Epoch:13 Training Loss: 0.49378055334091187, Training Accuracy: 86.96666717529297 in rank:4
Epoch:14 Training Loss: 0.32761284708976746, Training Accuracy: 88.70833587646484 in rank:0
Epoch:14 Training Loss: 0.25630053877830505, Training Accuracy: 88.83332824707031 in rank:1
Epoch:14 Training Loss: 0.43532446026802063, Training Accuracy: 88.61666870117188 in rank:3
Epoch:14 Training Loss: 0.270677775144577, Training Accuracy: 89.14167022705078 in rank:2
Epoch:14 Training Loss: 0.3893902897834778, Training Accuracy: 88.89167022705078 in rank:4
Test Loss: 0.34645663986206054,  Test Accuracy: 0.9302999973297119
Total Execution Time: 236.1258521080017
```

Again we see a decrease in the time as we increase the number of processes from 4 to 5.

We note down all these times and plot a speed up graph as follows:

```
processes=[1,2,3,4,5]
plt.plot(processes, [303.7632/303.7632, 303.7632/257.5364, 303.7632/236.5158, 303.7632/239.9027, 303.7632/236.1258], label='Hogwild', marker="p")
plt.title('SpeedUp Graph')
plt.xlabel('Number of Processes')
plt.ylabel('Parallel Efficiency')
plt.legend()
plt.show()
```



Overall we can see an increasing trend in the speed up graph.