Sharon Laurance

Implementing Parallel Stochastic Gradient Descent

Steps to implement parallel SGD in Pytorch.

1. Import the required libraries.
2. Partition the Data according to the number of workers. Assuming we have 2 replicas, then each process will have a train_set of 60000 / 2 = 30000 samples. We also divide the batch size by the number of replicas in order to maintain the overall batch size of 128. __init__ method is create partition in data and find out indexes for data.

```
""" Dataset partitioning"""
class Partition(object):

    def __init__(self, data, index):
        self.data = data
        self.index = index

    def __len__(self):
        return len(self.index)

    def __getitem__(self, index):
        data_idx = self.index[index]
        return self.data[data_idx]
```

```
class DataPartitioner(object):

    def __init__(self, data, sizes=[0.7, 0.2, 0.1], seed=1234):
        self.data = data
        self.partitions = []
        rng = Random()
        rng.seed(seed)
        data_len = len(data)
        indexes = [x for x in range(0, data_len)]
        rng.shuffle(indexes)

        for frac in sizes:
            part_len = int(frac * data_len)
            self.partitions.append(indexes[0:part_len])
            indexes = indexes[part_len:]

    def use(self, partition):
        return Partition(self.data, self.partitions[partition])
```

```
    """ Partitioning MNIST """
def partition_dataset():
    dataset = datasets.MNIST('./data', train=True, download=True,
                             transform=transforms.Compose([
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.1307,), (0.3081,))
                             ]))
    size = comm.Get_size()
    bsz = 128 / float(size)
    bsz=int(bsz)
    partition_sizes = [1.0 / size for _ in range(size)]
    partition = DataPartitioner(dataset, partition_sizes)
    rank = comm.Get_rank()
    partition = partition.use(rank)
    train_set = torch.utils.data.DataLoader(partition,
                                            batch_size=bsz,
                                            shuffle=True)
    return train_set, bsz
```

3. Define the Custom Model. It has 2 convolution layers, 3 max_pooling layers and 2 fully connected layers. ReLU is used for activation. It has a dropout layer to ensure that the model does not overfit. Output is log_Softmax which is used for NLL (Negative Log Likehood) Loss.

```
class Net(nn.Module):
    """ Network architecture. """

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(80, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(self.conv2_drop(F.max_pool2d(self.conv2(x), 2)))
        x=F.max_pool2d(x,2)
        x = x.view(-1, 80)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

4. Define the Training function – Optimizer used is SGD and learning_rate is 0.01.

```
""" Distributed Synchronous SGD"""
def run(rank, size):
    torch.manual_seed(1234)
    train_set, bsz = partition_dataset()
    model = Net()
    optimizer = optim.SGD(model.parameters(),
                          lr=0.01, momentum=0.5)
    num_batches = ceil(len(train_set.dataset) / float(bsz))
    rank = comm.Get_rank()
    for epoch in range(10):
        epoch_loss = 0.0
        running_acc=0.0
        for data, target in train_set:
            optimizer.zero_grad()
            output = model(data)
            loss = F.nll_loss(output, target)
            epoch_loss += loss.item()
            loss.backward()
            average_gradients(model)
            optimizer.step()
            running_acc += accuracy(output,target)
    print('Loss Rank ', rank, ', epoch ', epoch, ': ', epoch_loss / num_batches)
    print('Accuracy Rank ', rank, ', epoch ', epoch, ': ', running_acc / num_batches)
    return model
```

5. Define the function where model parameters are averaged over workers. It takes in a model and averages its gradients across all the workers.

```
""" Gradient averaging. """
def average_gradients(model):
    size = comm.Get_size()
    for param in model.parameters():
        comm.allreduce(param.grad.data, op=MPI.SUM)
        param.grad.data /= size
```

6. Define functions for calculating loss and accuracy.

```python
def validate(test_loader,model):
    '''
    Function for the testing step of the training loop
    '''
    model.eval()
    running_loss = 0
    running_acc=0
    for X, y_true in test_loader:
        # Forward pass and record loss
        y_hat= model(X)
        loss = F.nll_loss(y_hat, y_true)
        running_loss += loss.item() * X.size(0)
        running_acc += accuracy(y_hat, y_true)
    epoch_loss = running_loss / len(test_loader.dataset)

    return model, epoch_loss,running_acc
```

```python
def accuracy(output, y):
        pred_labels = torch.argmax(output, dim=1)
        return (pred_labels == y).sum().item() / len(y)
```

7. Define the main function to run the program. At rank==0 it calculates the test loss and test accuracy.

```python
if __name__ == "__main__":
    comm = MPI.COMM_WORLD
    rank = comm.Get_rank()
    size = comm.Get_size()
    model=run(rank, size)
    test_losses=[]
    test_dataset = datasets.MNIST('./data', train=False, download=True,
                                  transform=transforms.Compose([
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.1307,), (0.3081,))
                                  ]))


    test_loader = DataLoader(dataset=test_dataset,
                                      batch_size=64,
                                      shuffle=False)
    if rank==0:
        num_batches = ceil(len(test_loader.dataset) / float(64))
        for epoch in range(10):
            with torch.no_grad():
                model, test_loss,test_accuracy = validate(test_loader,model)
        print("Final Test Loss is",test_loss)
        print("Final Test Accuracy is",test_accuracy/num_batches)
        print("\n Time taken for {} workers = {}".format(size,round(MPI.Wtime()-start,4)))
```

Results:

```
(dda) C:\Users\Sharon>mpiexec -n 1 python "D:/Data Analytics/DDA LAB/Lab 10/file1.py"
Loss Rank  0 , epoch  9 :  0.1557514868390712 8
Accuracy Rank  0 , epoch  9 :  0.9542632818052594
Final Test Loss is 0.061746306309476494
Final Test Accuracy is 0.9804936305732485

 Time taken for 1 workers = 478.1132

(dda) C:\Users\Sharon>
```

```
(dda) C:\Users\Sharon>mpiexec -n 2 python "D:/Data Analytics/DDA LAB/Lab 10/file1.py"
Loss Rank  1 , epoch  9 :  0.21229579912097468
Accuracy Rank  1 , epoch  9 :  0.937311211798152
Loss Rank  0 , epoch  9 :  0.21277873056815633
Accuracy Rank  0 , epoch  9 :  0.9360785358919687
Final Test Loss is 0.08609042734727264
Final Test Accuracy is 0.9742237261146497

 Time taken for 2 workers = 191.6146
```

```
 Loss Rank  1 , epoch  9 :  0.21229579912097468
 Accuracy Rank  1 , epoch  9 :  0.937311211798152
 Loss Rank  0 , epoch  9 :  0.21277873056815633
 Loss Rank  1 , epoch  9 :  0.2529869018673147
 Accuracy Rank  1 , epoch  9 :  0.9244284715982873
 Loss Rank  0 , epoch  9 :  0.2545202533622708
 Accuracy Rank  0 , epoch  9 :  0.9231306778476636
 Final Test Loss is 0.11384102821797132
 Final Test Accuracy is 0.9656648089171974

  Time taken for 3 workers = 189.5582

 (dda) C:\Users\Sharon>
```

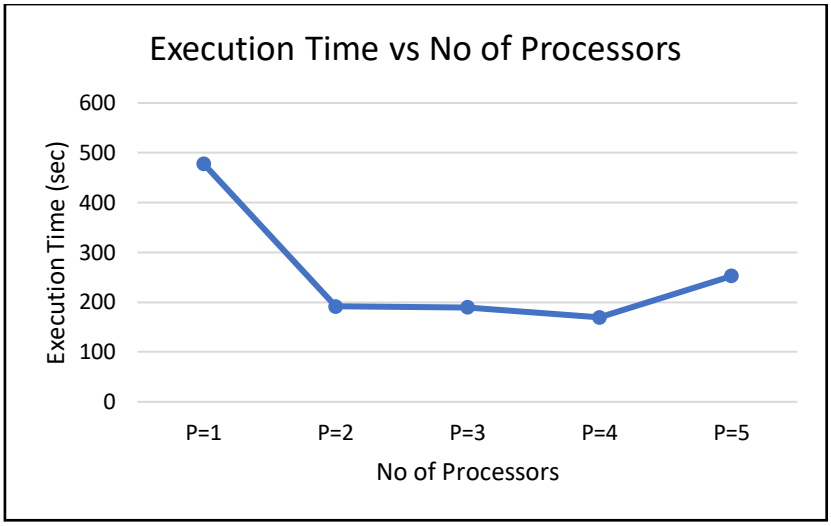
```
(dda) C:\Users\Sharon>mpiexec -n 4 python "D:/Data Analytics/DDA LAB/Lab 10/file1.py"
Loss Rank  1 , epoch  9 :  0.3020674043865219
Accuracy Rank  1 , epoch  9 :  0.9087375621890548
Loss Rank  3 , epoch  9 :  0.2956910384123895
Accuracy Rank  3 , epoch  9 :  0.9104477611940298
Loss Rank  2 , epoch  9 :  0.2935250889517859
Accuracy Rank  2 , epoch  9 :  0.9102478678038379
Loss Rank  0 , epoch  9 :  0.29351061727128813
Accuracy Rank  0 , epoch  9 :  0.90625
Final Test Loss is 0.1307129032254219
Final Test Accuracy is 0.9611863057324841

 Time taken for 4 workers = 169.6105
```

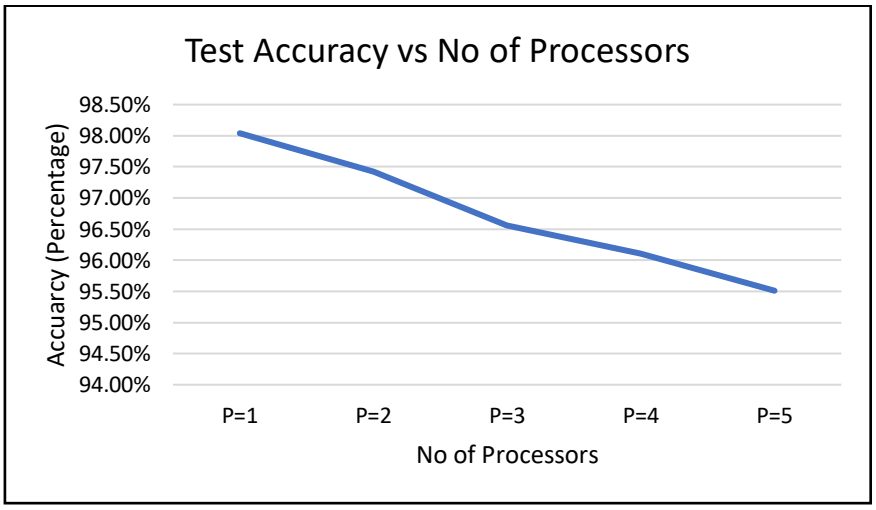
```
(dda) C:\Users\Sharon>mpiexec -n 5 python "D:/Data Analytics/DDA LAB/Lab 10/file1.py"
 Loss Rank  4 , epoch  9 :  0.33561893630928047
 Accuracy Rank  4 , epoch  9 :  0.8987499999999977
 Loss Rank  2 , epoch  9 :  0.3275257498801996
 Accuracy Rank  2 , epoch  9 :  0.9033333333333311
 Loss Rank  3 , epoch  9 :  0.315383940259926
 Accuracy Rank  3 , epoch  9 :  0.9051666666666643
 Loss Rank  1 , epoch  9 :  0.34103134986168393
 Accuracy Rank  1 , epoch  9 :  0.896416666666665
 Loss Rank  0 , epoch  9 :  0.3383554055821151
 Accuracy Rank  0 , epoch  9 :  0.8957499999999979
 Final Test Loss is 0.15667753113508223
 Final Test Accuracy is 0.9551154458598726

  Time taken for 5 workers = 252.7477
```

No of Workers	P=1	P=2	P=3	P=4	P=5
Execution Time (sec)	478.11	191.61	189.55	169.61	252.74
Accuracy	98.04%	97.42%	96.56%	96.11%	95.51%



Comment: We can see the execution time is decreasing due to parallelization. But as the machine is a 4-core machine at P=5, we can see the time increasing due to communication overhead. Parallel SGD can help with Machine learning speedup, but only by itself, the improvement is not quite significant. If work with model and data parallelization, that one can achieve better performance.



Comment : We can see the test_accuracy decreases but the decreases for 5 processors is 96% accuracy. There is slight decrease.

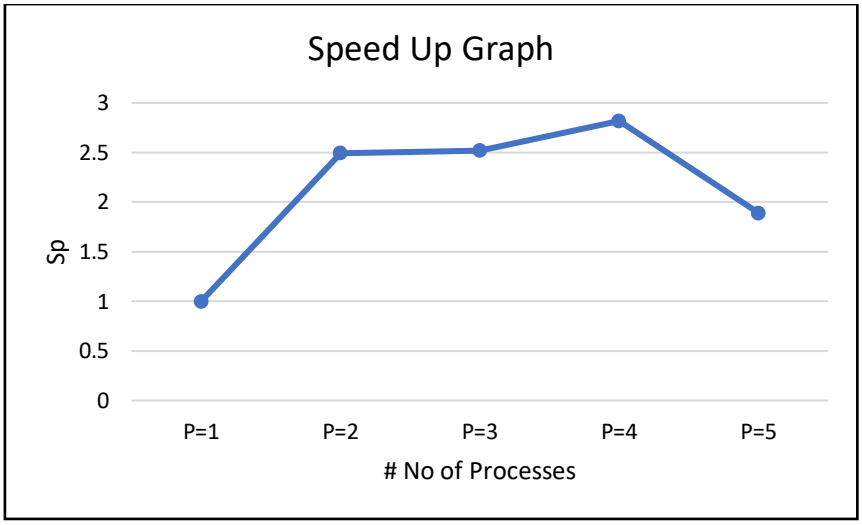


Fig: Speed Up Graph. We can see a sublinear trend for Speed Up graph of PSGD. Adding workers helps to speed up the execution of the MNIST Classification till P=4. After that due to communication overhead overall speed decreases. There is tradeoff between computation cost and communication cost.

PyTorch distributed execution

HogWild SGD – It leverages parallel processes on computers to run SGD simultaneously and asynchronously. Asynchronicity is important for reducing unnecessary idle time in which no calculations are done but energy is still consumed.

Using Hogwild, each participating process in a parallel training session is responsible for one partition of the data, e.g. having 8 parallel processes would split the data set in 8 equal parts whereas each process is assigned to one part. Moreover, on shared memory or a separate server, an initial model is created which can be accessed by all processes.

Once the training starts, each process loads the current state of the model from shared memory and starts reading the first batch of their data partition. As in standard SGD, each process is calculating the gradients for that batch. The gradients are now directly written to the shared model without blocking the other processes. Once written, the new model parameters are loaded and the next batch is used. Due to the missing blocking, the shared model will sometimes receive old gradients which, one may think, could be a disadvantage. The research shows that the training even benefits from the non-blocking fashion.

Steps for Implementing HogWild

1. Import required Libraries

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

2. Define the Model. The same model architecture is used as in Question 1. The model gives back the LogSoftmax which is useful when using NLL_Loss during the training.

```
class Model(nn.Module):
    """ Network architecture. """

    def __init__(self):
        super(Model, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(80, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(self.conv2_drop(F.max_pool2d(self.conv2(x), 2)))
        x=F.max_pool2d(x,2)
        x = x.view(-1, 80)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)
```

3. Define the test_accuracy() function

```
def test_accuracy(model, data_loader):
    print("Test started...")
    correct = 0
    total = 0
    with torch.no_grad():
        for data, labels in data_loader:
            output = model(data)
            _, predicted = torch.max(output.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy of the network on the test images : %d %%' % (100 * correct / total))
```

4. Define train model. Train_model function, is taking a model and a data loader as input parameters. Inside of this function, the Adam optimizer is used and mentioned NLLLoss. Each process participating in Hogwild will call it at the same time.

```
def train_model(model, data_loader):
    optimizer = optim.Adam(model.parameters())
    criterion = nn.NLLLoss()

    for data, labels in data_loader:
        optimizer.zero_grad()
        loss = criterion(model(data), labels)
        loss.backward()
        optimizer.step()
```

5. Define the main function. We define the number of parallel processes, instantiate the model and push it to shared memory with the single method call share_memory(). The dataset used is the MNIST, available in the torchvision package.
   It loops over the processes and define a data loader for each process. The data loaders hold a distributed sampler which knows the rank of the process and handles the distribution of the data. Hence, each process has its data partition. The multiprocessing package calls the training function within each process and waits, with the join command, for the processes to finish. During training, all the processes have access to the shared model but train only on their very own data partition. Thus, we decrease the training time by the number of training processes.

```
: > Data Analytics > DDA LAB > Lab 10 > file3.py > ...
58  if __name__ == '__main__':
59      num_processes = 4
60      model = Model()
61      model.share_memory()
62      dataset = datasets.MNIST('./data', train=True, download=True,
63                               transform=transforms.Compose([
64                                   transforms.ToTensor(),
65                                   transforms.Normalize((0.1307,), (0.3081,))]))
66      testset = datasets.MNIST('./data', train=False, download=True,
67                               transform=transforms.Compose([
68                                   transforms.ToTensor(),
69                                   transforms.Normalize((0.1307,), (0.3081,))]))
70      processes = []
71      for rank in range(num_processes):
72          data_loader = DataLoader(
73              dataset=dataset,
74              sampler=DistributedSampler(
75                  dataset=dataset,
76                  num_replicas=num_processes,
77                  rank=rank
78              ),
79              batch_size=32
80          )
81          p = mp.Process(target=train_model, args=(model, data_loader))
82          p.start()
83          processes.append(p)
84      for p in processes:
85          p.join()
86      test_accuracy(model, DataLoader(
87          dataset=testset,
88          batch_size=1000))
89      print("\n Time taken for {} processes is {}".format(num_processes,round(time.time()-start_time,4)))
```

Results:

```
(dda) C:\Users\Sharon>python "D:/Data Analytics/DDA LAB/Lab 10/file3.py"
Test started...
Accuracy of the network on the test images: 92 %

 Time taken for 1 processes is 32.4099
```

```
(dda) C:\Users\Sharon>python "D:/Data Analytics/DDA LAB/Lab 10/file3.py"
Test started...
Accuracy of the network on the test images: 92 %

 Time taken for 2 processes is 21.4979
```

```
(dda) C:\Users\Sharon>python "D:/Data Analytics/DDA LAB/Lab 10/file3.py"
Test started...
Accuracy of the network on the test images: 93 %

 Time taken for 3 processes is 18.3385
```

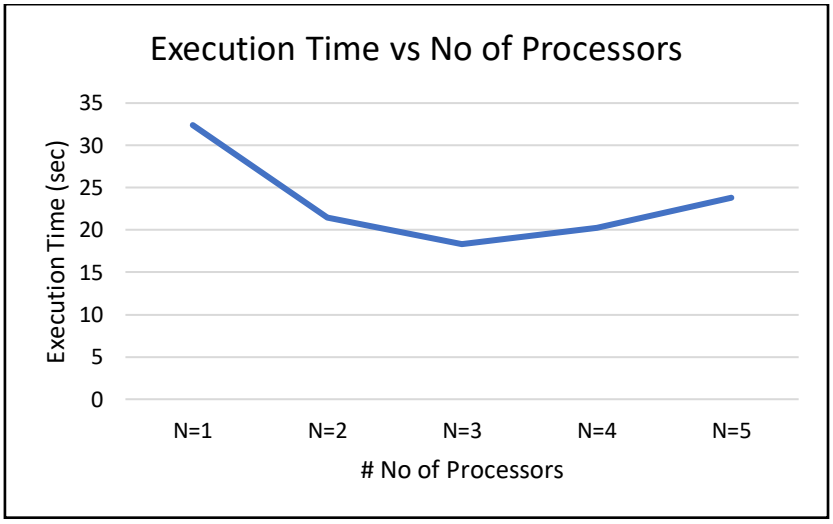
```
(dda) C:\Users\Sharon>python "D:/Data Analytics/DDA LAB/Lab 10/file3.py"
Test started...
Accuracy of the network on the test images: 92 %

 Time taken for 4 processes is 20.2208
```

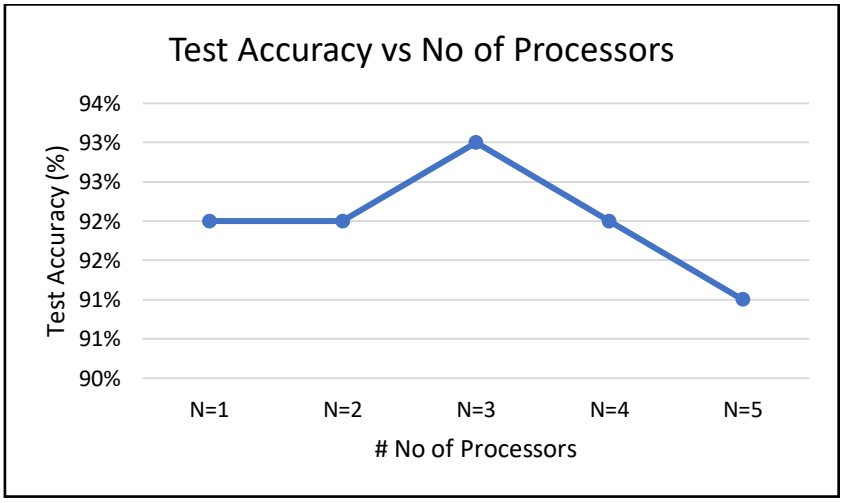
```
(dda) C:\Users\Sharon>python "D:/Data Analytics/DDA LAB/Lab 10/file3.py"
Test started...
Accuracy of the network on the test images: 91 %

 Time taken for 5 processes is 23.8279
```

No of Processors	N=1	N=2	N=3	N=4	N=5
Execution Time (sec)	32.4	21.49	18.33	20.22	23.82
Test Accuracy	92%	92%	93%	92%	91%



Comment: We can see the execution time is decreasing due to parallelization. But as the machine is a 4-core machine at P=5, we can see the time increasing due to communication overhead.



Comment : We can see the test_accuracy increases till P=3 but the decreases from P=4 processors. It increases from 92 to 93% but at 5 processors it goes to 91%.

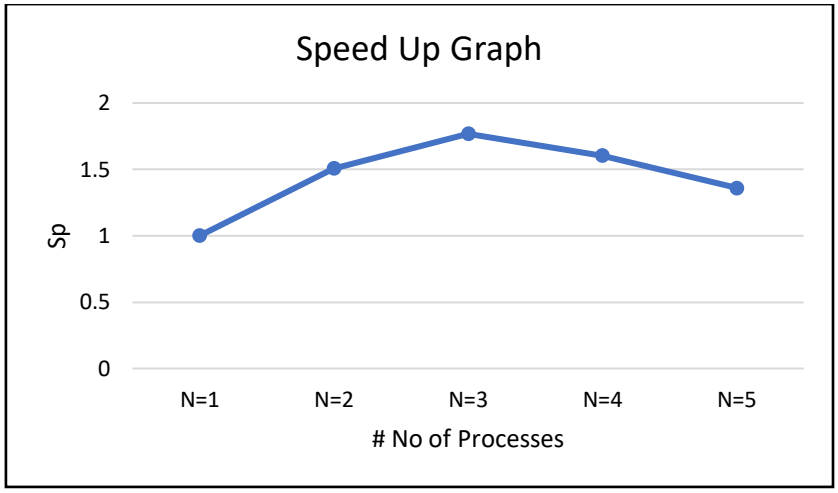


Fig: Speed Up Graph. We can see a sublinear trend for Speed Up graph of HogWild SGD. Adding workers helps to speed up the execution of the MNIST Classification till N=4. After that due to communication overhead overall speed decreases. There is tradeoff between computation cost and communication cost.

References:

[1]https://pytorch.org/tutorials/intermediate/dist_tuto.html#:~:text=The%20following%20steps%20install%20the,python%20setup.py%20install%20yet.&text=Now%2C%20go%20to%20your%20cloned,execute%20python%20setup.py%20install%20.

[2] https://cse.buffalo.edu/faculty/miller/Courses/CSE633/Jiarui-Liu-Spring-2021.pdf

[3] https://pytorch.org/docs/stable/distributed.html#backends-that-come-with-pytorch

[4] https://yuqli.github.io/jekyll/update/2018/12/17/sgd-mpi-pytorch.html

[5] https://towardsdatascience.com/this-is-hogwild-7cc80cd9b944

[6] https://github.com/wenig/hogwild/blob/master/index.py