Parallelizing Deep Neural Networks using MPI and GPU Computing

Final Project Report for COMP 605, Spring 2018

Pradeep Singh

Introduction:

In last few years, the field of Machine Learning has grown exponentially and has produced some state of the art results in computer vision and natural language processing etc. area. The part of this progress comes from deep neural network based models, also called deep learning/ deep neural nets.

Deep learning models are very powerful because of their ability to learn high and low level representation of the data (image, voice, language, etc.), but this comes at cost of large training time and mind boggling compute requirement. This is largely because any deep learning model is grounded on learning parameters over the entire parameter space hundreds of times. And, this process is nothing but large matrix multiplication operations. Computing these large matrices takes lot of time and power. Thus, in last few years GPU computing have provided an viable option that could speed up the entire learning/ training. This project explores a basic parallel deep neural network models and apply them over Fashion MNSIT dataset. I have developed two version of this model using Python – sequential and parallel version.

In this project, I have implemented a basic neural network model in Python using MPI and GPU computing. The goal of this project is to develop a sequential and parallel neural network model using MPI and GPU and measure the improvement in performance and speed up in training time. I have explored various methods to parallelize a neural network model -- data based parallelism, model based parallelism etc. but have only implemented data based parallelism.

Neural Network:

Neural networks are computing systems that are loosely inspired by the biological neurons. They are the building blocks of Deep learning models. In last decade or so, they have completely revolutionized the machine learning. Intuitively, neural networks are the nothing but the nonlinear function, which passes non-linearity to the output as input propagates through them. Any neural network is consist of multiple layers, which all the layers can have different number of neural units. When these layers are stacked on top of each other, increasing the depth of the model thus we call them deep neural nets. All these layers are connected to the previous/ next layer through weights.

Each layer learns some high/ low level representation of the data. Thus, by adding more layers we can learn more abstraction of the data and thus leading to better performance. But, adding more layers increases the complexity of model and thus leads to high compute time and power.

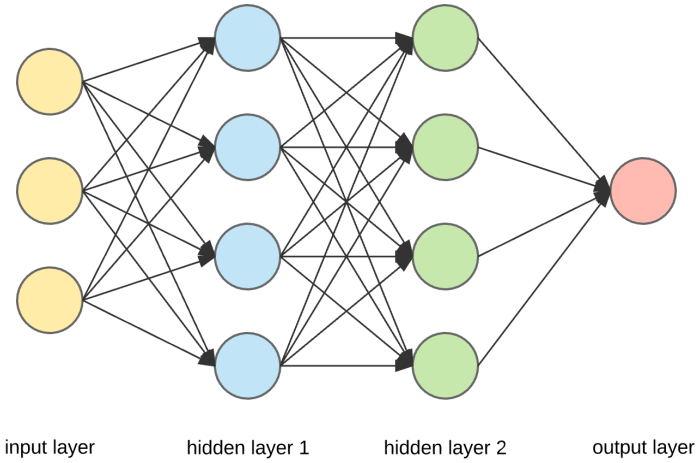

Figure 1: Neural Network with 2-hidden layers

Neural networks are computationally intensive, because they need to update millions of parameters numerous times in order to minimize error and produce an accurate model. Those updates are basically nothing but large matrix multiplication operations. The only way to train neural networks on very large datasets is to give them lots of time, lots of cores, or both.

Dataset:

The Fashion-MNIST is a dataset of Zalando's articles images – consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each examples is 28X28 grayscale image, associated with a label from 10 classes.

Target Class	Definition
0	T-shirt/ Top
1	Trouser
2	Pullover
3	Dress
4	Coat
5	Sandal
6	Shirt
7	Sneaker
8	Bag
9	Ankle Boot

Figure 2: Fashion-MNIST dataset – Number of classes with labels

The dataset was divided into 3 parts – training set, validation set and testing set. All the different versions of different models were trained on training data first and then were evaluated using validation set. Only the best models were used to test the testing data set.

Parallelism: Model and Data based

Neural networks based models can be parallelized in two ways: Model based parallelism and Data based parallelism. I have specifically focused on data based parallelism method.

Data parallelism:

Data parallelism is basically when you use the same model for every thread, but feed it with different parts of the data and model parallelism is when you use the same data for every thread, but split the model among threads. For neural networks this means that data parallelism uses the same weights and but different mini-batches in each thread and the gradients need to be synchronized, i.e. averaged, after each pass through a mini-batch.

Model parallelism:

Model parallelism splits the weights of the net equally among the threads and all threads work on a single mini-batch; here the generated output after each layer needs to be synchronized, i.e. stacked, to provide the input to the next layer.

Parallelizing the neural network in this way (either of the methods) can significantly reduce the training time for the model. And, this coupled with GPU computing can even speed up the process. I have used MPI and GPU computing technologies are used to implement data parallelism. The core concept is to split the input dataset, distribute the subsets to the worknodes, and collect the results computed by GPU.

MPI: Message Passing Interface

MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be.

MPI primarily addresses the message-passing parallel programming model: data is moved from the address space of one process to that of another process through cooperative operations on each process.

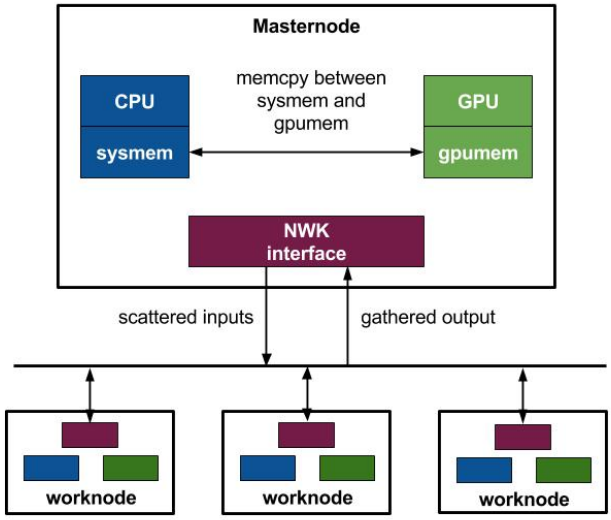

Figure 3: Concept of data parallelism on GPU cluster

MPI standard provides many operations that can be used to distribute the workload/ task parallelly among different processes in the same machine or in a cluster. Some of the operations that I have used in this project are: broadcast, scatter and gather. I have specifically used MPI for Python package (mpi4py) to execute above mentioned MPI operation in my model.

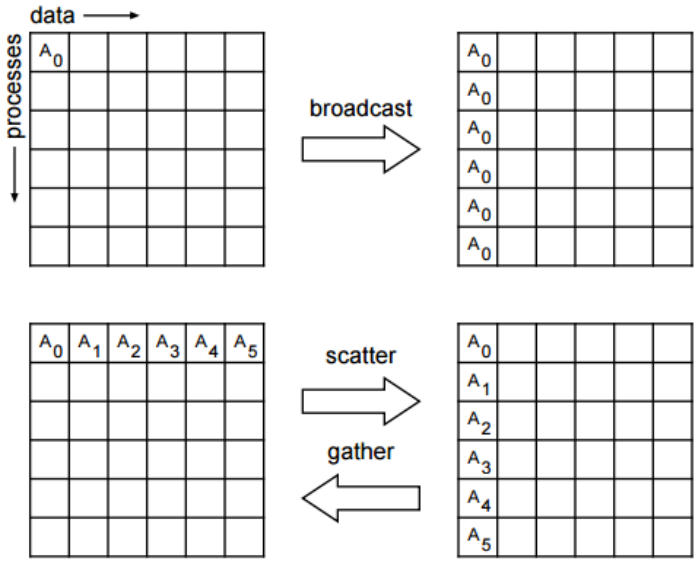

Figure 4: Some of MPI collective functions

To distribute dataset to worknodes (Scatter) in mpi4py. The scatter function splits a dataset into several subsets and makes the subsets evenly distributed to a group of processes,

```
sliced_inputs = np.asarray(np.split(inputs, num_worknodes))
sliced_labels = np.asarray(np.split(labels, num_worknodes))
inputs_buf = np.zeros((len(inputs)/num_worknodes, Input_layer_size))
labels_buf = np.zeros((len(labels)/num_worknodes))
comm.Scatter(sliced_labels, labels_buf)
```

To collect training results of worknodes (Gather) in mpi4py. the gather function does the opposite, collects data from a group of processes, and reassembles the collected data into a complete dataset,

```
cost, (theta1_grad, theta2_grad) = cost_function(
    theta1, theta2,
    Input_layer_size,
    Hidden_layer_size,
    Output_layer_size,
    inputs_buf, labels_buf,
    regular=0)
theta1_grad_buf = np.asarray([np.zeros_like(theta1_grad)] * num_worknodes)
comm.Gather(theta1_grad, theta1_grad_buf)
theta1_grad = functools.reduce(np.add, theta1_grad_buf) / num_worknodes
```

To synchronize the weights of neural network among worknodes (Bcast) using mpi4py. The broadcast function helps a process share information with the other processes. All the processes will get a copy of the data assigned by the broadcast function,

```
theta1 -= learningrate * theta1_grad
comm.Bcast([theta1, MPI.DOUBLE])
```

Matrix Multiplication on GPU using Theano:

Because there are a lot of matrix multiplications involved in the training process, it will be good to use GPU to speed up the computation. I have used a library called Theano. Theano provides high level functions for matrix based operations on GPU. Here is the code using Theano to multiply two matrices:

```
def gpu_matrix_dot():
    x = T.matrix('x')
    y = T.matrix('y')
    z = T.dot(x, y)
    f = theano.function(
        [x, y], z)
    return f

Matrix_dot = gpu_matrix_dot()
c = Matrix_dot(a, b)
```

Environment:

The below mentioned libraries and packages were used for this project. I have trained all my models on Google cloud.

1. Virtual CPU: (n1-standard-1)
2. Intel i7-5930 3.5 GHz
3. System memory: 3.75 GB
4. GPU: Nvidia Titan X
5. NumPy: For matrix operations in CPU mode
6. Theano: For matrix operations in GPU mode
7. mpi4py: MPI for Python package
8. Python 3.6.5

Experiments:

In order to measure speed up, I have tried different experiments with different models on different hardware (CPU and GPU). I have trained my parallel and sequential neural network models on CPU and GPU.

- Data parallelism vs Sequential execution: with CPU
- Data parallelism vs Sequential execution: with GPU

Any typical neural network model typically comprises of many parameters which needs to be optimized. I have tried various combinations of parameters and have reported just the one that worked best for my model.

The parameters of the learning algorithm:

- Learning rate: 0.1
- Gradient descent iteration: 60
- Activation function: sigmoid

Figure below show the results of data parallelism performance comparison between sequential and parallel models.

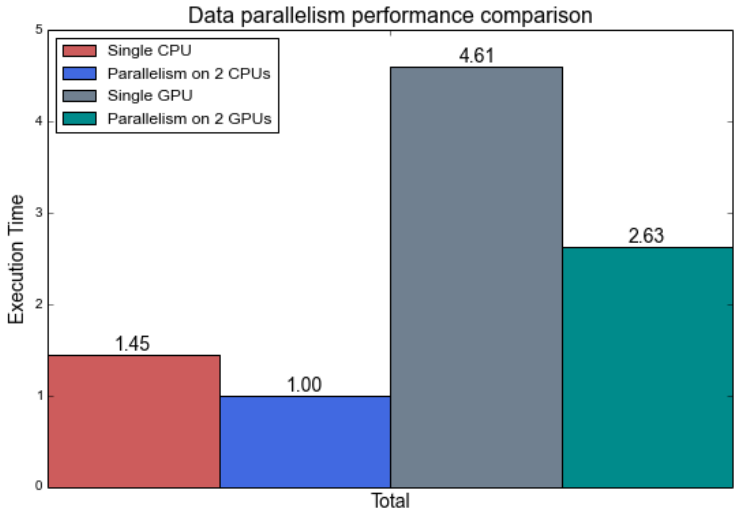

Figure 5: Data parallelism performance comparison

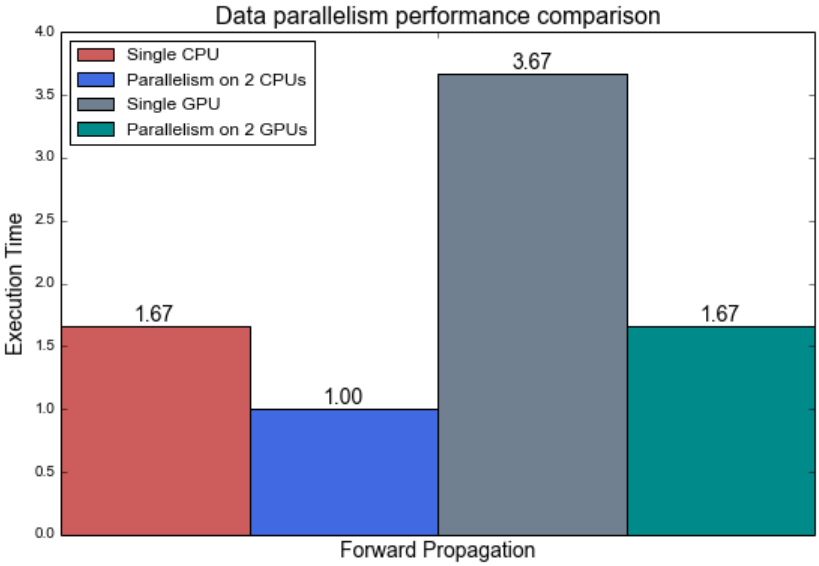

Figure 6: Performance comparison at forward propagation stage

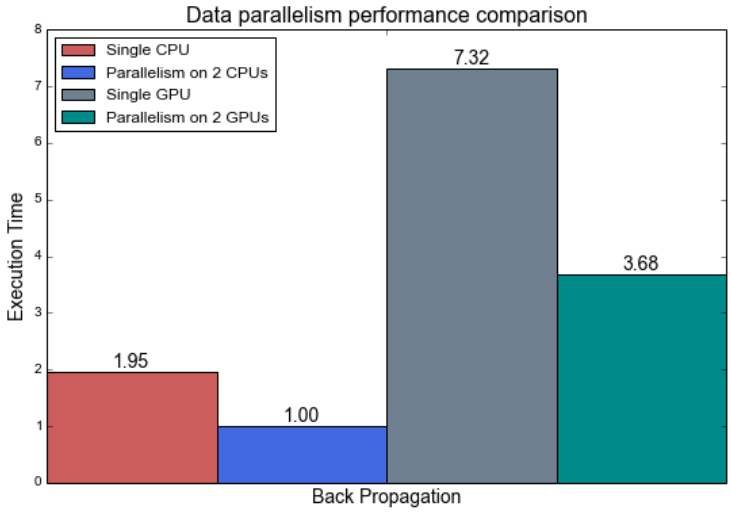

Figure 7: Performance comparison at forward propagation stage

Observations:

1. Data parallelism brings performance improvement.

We are getting some improvement but at the first glance, the improvement seems to be trivial. If the dataset is split and distributed evenly on two machines (so doubled computing units), of course the speed can also be two times faster than using a single machine alone. However, the really important fact behind is that we are now able to scale-out the computing ability from one machine to the GPU cluster easily. Since GPU computing is still one of the most effective technology for deep learning, this technique can make the learning process much faster.

2. GPU computing seems to be slower than CPU computing, is it true?

The fact that GPU is slower in above figures does not mean that CPU is a better tool than GPU for deep learning. Actually, it just reflects the fact that the cost of memory copy between system and GPU memories is expensive. Because the size of the sample neural network is small, the computation loads are not really heavy. Therefore, the improvement to the total computing time by using GPU is overwhelmed by the time consumed while copying inputs and outputs between system memory and GPU memory.

When the size of the data is bigger, the ratio of time to compute and time to execute memory copy is also getting larger; GPU computing starts to show its power. Huge dataset and larger number of hidden layers make GPU computing a desirable tool for deep learning.

Conclusion:

Among all experiments (CPU vs GPU), we can see that our parallel model is getting more speedup as compared to sequential model. On CPU based computing, we are getting nearly 50% speedup and on GPU based computing we are getting more than 50% speedup. This is in line with what is expected when you distribute your workload and parallelize your learning/ training process.

I believe we can achieve more speed up if we use large dataset and or have a more complex/ deep model. GPU computing can also be exploited much more by using CUDA.