A Massively Parallel Artificial Deep Neural Network Framework for IBM Blue Gene/Q

Mukul Surajiwale, Rabiul Chowdhury, Kyle Fawcett, Alex Giris
CSCI 4320: Parallel Programming Spring 2017
Rensselaer Polytechnic Institute
surajm@rpi.edu, chowdr@rpi.edu, fawcek@rpi.edu, girisa@rpi.edu

***Abstract*— Ninety percent of all data generated to date has been generated in the past two years [2]. The tremendous growth of the Internet has made it easy to collect data and obtain valuable insights. An equally rapid growth in computing ability has led to a renewed interest in using machine learning techniques to obtain insights from data. More specifically, artificial deep neural networks have emerged as the most promising model to perform a variety of tasks such as image recognition and autonomous driving. However, ANNs (Artificial Neural Networks) are very computationally heavy and it is well known that ANNs do not scale well as the complexity of the model increases. In this paper we propose an easy to use massively parallel artificial deep neural network framework for the IBM Blue Gene/Q High Performance Computing (HPC) architecture. We first provide a brief introduction to the structure of neurons in the human brain and how it relates to the design of an ANN. We then provide a detailed description of our framework and our use of OpenMPI to introduce parallelism. Lastly, we conclude by providing an analysis of our performance results which show that by increasing the amount processes our framework is able to achieve significant speedups in training.**

I. INTRODUCTION

A. Overview of Neurons in Human Brain

The design of the Artificial Neural Network model is loosely inspired by the human brain. The human brain is composed of nearly 100 billion neurons that each contain approximately 1000 synapses [4]. Each neuron in composed of a soma and an axon. Additionally, neurons contain dendrites. Each neuron in the brain maintains a constant voltage gradient. If the voltage gradient is changed due to some electrochemical process, the neuron releases an electrical impulse that travels down the neurons axon. The electrical impulse is then transmitted to all its neighbor neurons that are connected to the sender neuron through its synapses. The electrical signal continues to travel in a similar manner through the neighbor neurons and their neighbors stopping once it has lost its electrical potential falls below a certain threshold. The connections formed between neurons are not static. Connections change overtime and often become stronger upon repeatedly doing a certain task. The processes through which neurons create, modify, and destroy their connections is what allows humans to perform a variety of non-trivial tasks such as learning to walk, recognizing objects, and speaking multiple languages.

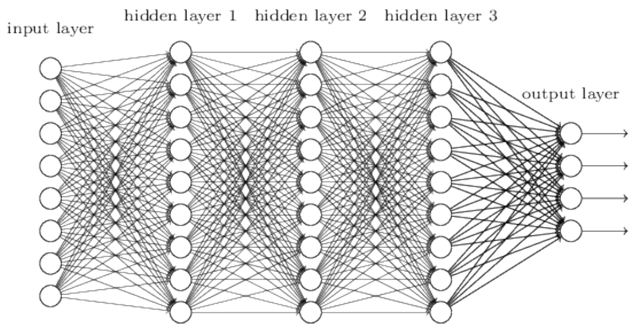

Fig. 1. Representation of a deep neural network with 8 input neurons, 9 neurons in each hidden layer, and 4 output neurons [5].

B. Overview of ANN Structure

Similar in design to the human brain, the Artificial Neural Network model is composed of a large network of artificial neurons that are connected to each other. More specifically, the type of ANN we focus on is called a Multilayer Perceptron (MLP), which is also the most commonly used ANN configuration. A single perceptron is the simplest ANN model which consists of multiple input neurons and a single output neuron. The input neurons transfer their input values to the output neuron via connection called a weight. The output neuron performs a summation of linear combination of the values from the input neurons and their respective weights. Lastly, the resulting value is passed through a non-linear activation function to threshold the output value.

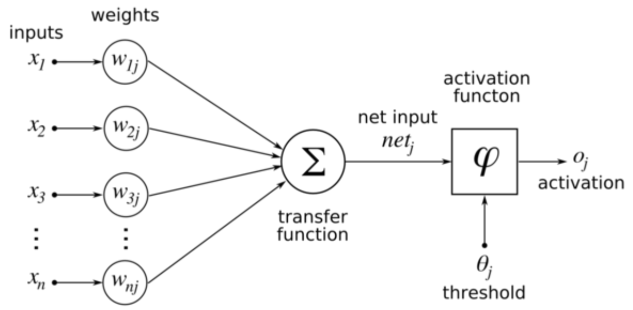

Fig. 2. Representation of a perceptron. Transfer function performs summation of input values and weights [6].

As the name suggests, a multilayer perceptron is composed of many perceptrons ”stacked” on top of each of each other to form a layer that is often referred to as a hidden layer. A deep neural network is composed of multiple hidden layers. The magic of deep neural network is their ability to approximate non-linear functions and thus perform many complex tasks such as music generation. Neural networks do not have to be explicitly programmed to perform a particular task. Instead given sufficient data, neural networks iteratively learn to approximate the underlying non-linear function of the given task. During the learning process input values are propagated through the network to calculate and estimated output. The estimated output value is compared with the expected output values and an loss value is calculated. The loss value is then reverse propagated through the network during which the weights are adjusted. This process repeats for thousands of iterations until the loss value converges.

C. Scaling ANNs

There are several issues with scaling neural networks to large sizes. One of these issues is the plasticity issue, or the issue of which connections should be hardwired and which should be "plastic". As the number of nodes in the network increases, there is an exponential increase in the number of possible networks. This issue also applies to the weight of each connection. As the number of connections increases, there is an exponential increase in the number of possible combinations of how each connection is weighted. For obvious reasons, this greatly increases the time needed to train larger networks. Typically, there is a cubic rate of increase in the amount of time needed to train a network as the number of nodes increases. For this reason especially, it is prohibitively expensive to construct artificial neural networks on a very large scale. [7]

II. TRAINING ALGORITHM OVERVIEW

In this section we will briefly describe the training process for a multilayer perceptron. The process consists of three steps called forward propagation, backward propagation, and updating the weights.

A. Variables

X = Input vector
W_i^L = ith column of the the weight matrix of the Lth layer
W_0^L = bias vector of the Lth layer
H^L = Lth hidden layer
$Y^{'}$ = predicted output
Y = expected output
ΔY = gradient of Y

B. Forward Propagation

In the forward propagation step input values are propagated from the input layer to the output layer. Each neuron in a hidden layer acts like a perceptron and performs a summation of the linear combination of the values from the neurons in the previous layer. The values of the bias neurons are also added in this step. Lastly, the resulting value is passed through an non-linear activation function. There are many different activation function such as ReLU, tanh, and Sigmoid. Artificial neural networks for multi-class classification often use the Sigmoid activation function.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{1}$$

Sigmoid activation function.

$$H^L = \sigma((W^L)^T H_i^{L-1} + W_0^L) \tag{2}$$

Calculate the output output layer.

In multi-class regression the output layer contains a Softmax activation function instead of the traditional Sigmoid function.

$$Y_i = \frac{e^{z_i}}{\sum_{k=1}^{K} e^{z_k}} \tag{3}$$

Softmax function for last layer. Y_i is the output of the ith neuron in the output layer, K is the number if classes/output neurons, and z_k is the input to the kth output neuron.

The final step of forward propagation is to calculate the gradient of the output Y and the loss given the predicted output and expected output.

$$Loss(Y^{'}, Y) = \frac{1}{2}(Y^{'} - Y)^2 \tag{4}$$

Squared loss function.

$$\Delta Y = \frac{\partial Loss(Y^{'}, Y)}{\partial Y} = Y^{'} - Y \tag{5}$$

Gradient of the output.

C. Backward Propagation

During the backward propagation step the error produced by the forward propagation step is propagated through the network. During this process the weights of network are adjusted with the goal of reducing the loss in the next iteration. Using ΔY one can calculate ΔW^L, ΔW_0^L, and ΔH^L.

$$\Delta W^L = \frac{\partial Y^{'}}{\partial W^L} \Delta Y \tag{6}$$

$$\Delta W_0^L = \frac{\partial Y^{'}}{\partial W_0^L} \Delta Y \tag{7}$$

$$\Delta H^L = \frac{\partial Y^{'}}{\partial H^L} \Delta Y \tag{8}$$

D. Gradient Descent

The final step is to update the weights of the network using a learning rate η. The learning rate is a scaling factor that determines the magnitude in which the weights are updated. A high learning rate will cause the weights to update more drastically. Although this can lead to faster convergence, it does not lead always lead to the optimal solution and can even cause the loss to increase over time. The process of updating the weights is known as gradient descent.

$$\Delta W^L(t) = W^L(t-1) - \eta \Delta W^L \tag{9}$$

Weight update rule where t is the weight values at the current iteration.

E. Performance

Deep neural networks are often seen as black boxes. It is very hard to tell exactly what the net is learning in any given iteration. There is no set formula that given a task outputs the number of hidden layer, number of neurons in each hidden layer, and the number of iterations required to learn the task. As a result, some people often say training a neural network is half art and half science. However, one thing that is known for sure is that in neural nets the forward, backward propagation, and gradient descent steps do not scale well. It takes $O(n)$, where N is the number of weights, for each iteration to execute. Thus, scaling an ANN model is very taxing on execution speed.

III. Our Approach

In this section we will explain how we approached the problem of reducing the computation time by introducing parallelism. We will first provide a brief overview of related work. We will then describe the design of the framework followed by an explanation of the parallel forward and backward propagation algorithms.

A. Related Work

Due to the immense popularity of neural networks and the growing demand for larger and more complex models, there are several paper describing parallel configurations for reducing the training time for a network. Since we do not have an extensive background in neural networks we have chosen to base our design on the one created by Lyle Long and Ankur Gupta of Pennsylvania State University [2].

Other related work includes a paper by George Dahl et al. [8]. In the paper, the writers present a technique for parallelizing neural networks on a cluster of workstations. They provide a solution called Pattern Parallel Training to counter the high network latencies and low bandwidth of workstation clusters. Their method involves duplicating the full neural network at each cluster node, where a subset of the traning set is trained by the processes in the cluster.

B. Framework Design

Our goal in designing the framework was to make it as easy as possible to quickly setup a network model that is both robust and one that can be parallelized quickly. The framework is composed of the Network, Layer, and Neuron classes. The network class manages the entire network and serves as a high level abstraction for the developer to use when constructing their model. The Network class is what manages each layer in the network. The Layer, as the name suggests, represents a single Layer in the network. A Layer can be of the "input", "hidden", "output" types. The Layer class also holds a vector of Neuron objects that represent the neurons in the layer. Each layer can have any number of neurons. Lastly, the Neuron class represents an individual neuron object. Each Neuron object keeps track of its current output, error, gradient, and a list of all its outgoing connections and their respective weights.

Listing 1. Creating and training a simple ANN

```
Network net = Network();
net.addLayer("input", 3);
net.addLayer("hidden", 256);
net.addLayer("hidden", 512);
net.addLayer("output", 128);
net.initializeNetwork();

net.loadTestingInputData("testInput.txt");
net.loadTestingOutputData("testLabels.txt");

int iterations = 1000;
for (int i = 0; i < iterations; i++) {
        net.forwardPropagation();
        net.computeLoss();
        net.backwardPropagation();
}
```

C. Parallel Configuration

In order to reduce the computation time of the neural network we chose to use OpenMPI to facilitate message passing to parallelize the network structure. Traditional neural networks are fully connected meaning that each neuron is connected to each neuron in the following layer. If we were to parallelize the fully connected nature of the neural network we would have to bear substantial communication overhead. Thus, we have chosen to only communicate the output values of neurons on each rank boundary. More specifically, figure 3 shows our use of ghost neurons to capture the output values of neurons from a different rank. By only communicating the output values of the neurons on the rank boundaries we are able reduce the communication overhead in the network. When running the network in parallel the neurons in each layer other than the input are distributed equally to each rank. Thus, no one rank is given a heavier computational load than the other. The MPI functions listed below are what we use to facilitate message passing.

- MPI_Isend and MPI_Irecv to pass the output values of the rank boundary neurons to the ghost neurons of the appropriate ranks.
- MPI_Allgather to gather the final combined ouput values in order to compute the loss.
- MPI_Barrier to synchronize all ranks.

IV. PARALLEL ALGORITHM

In this section we provide an overview of our implementation of the forward propagation algorithm which we have modified to support message passing to allow parallelization. The figure 3 shows a network with 3 input neurons, 9 hidden neurons, and 6 output neurons running with 4 ranks where ranks 0 is the master rank. It should be noted that the input is fed into each rank.

A. *Forward Propagation*

During forward propagation input values are propagated through the network to produce an expected output. In order to parallelize the process we use OpenMPI to pass messages between each of the ranks. In our framework rank 0 is the master rank which gathers the output from each rank and compute the joint loss value. The remaining ranks execute the algorithm starting from the first hidden layer. First, each layer will request all output values from the neurons from the previous layer. Then, with these values, each neuron in the current layer will calculate its output value by performing a summation over all its incoming weights and their respective neuron's output value. Lastly we pass the resulting values though the Sigmoid activation function to generate the neuron's final output.

Once the outputs for each neuron in the layer are calculated we begin the message passing step. Each layer contains a *ghostTop* and *ghostBottom* neuron used to hold the values received from other ranks. Each rank's task of determining which rank to send its values to and from which it expected to receive values results in one of the following three cases.

- **Case 1:** In this case the first rank is executing forward prop on the first n number of neurons in a hidden layer l where $n =$ (total number of neurons in hidden layer) $/(workingranks)$. In this case the rank sends the output of its first neuron to the *ghostBottom* neuron of the last rank. Additionally, the output of its last neuron is sent to the *ghostTop* neuron of rank 2. Next, the current rank's *ghostBottom* neuron will receive the output of the first neuron from the rank 2. Finally, the currents rank's *ghostTop* will receive the output of the last neuron from the last rank.
- **Case 2:** In this case the last rank is executing forward prop on its portion of the hidden layer. The rank first sends the output of its first neuron to the *ghostBottom* neuron of the rank before it. Then it sends the output of its last neuron to the *ghostTop* neuron of the first rank. Next, the current ranks *ghostBottom* neuron will receive the output of the first neuron from the first rank. Finally, the current rank's *ghostTop* will receive the output of the last neuron from the rank before it.
- **Case 3:** In this case the rank is neither the first one or the last one. Thus, this case is applied to all ranks in between. The rank first sends the output if its first neuron to the *ghostBottom* neuron of the rank before it. Then it sends the output if its last neuron to the *ghostTop* neuron of the rank after it. Next, the current ranks *ghostBottom* neuron will receive the output of the first neuron from the rank after it. Finally, the current ranks *ghostTop* neuron will receive the output of the last neuron from the rank before it.

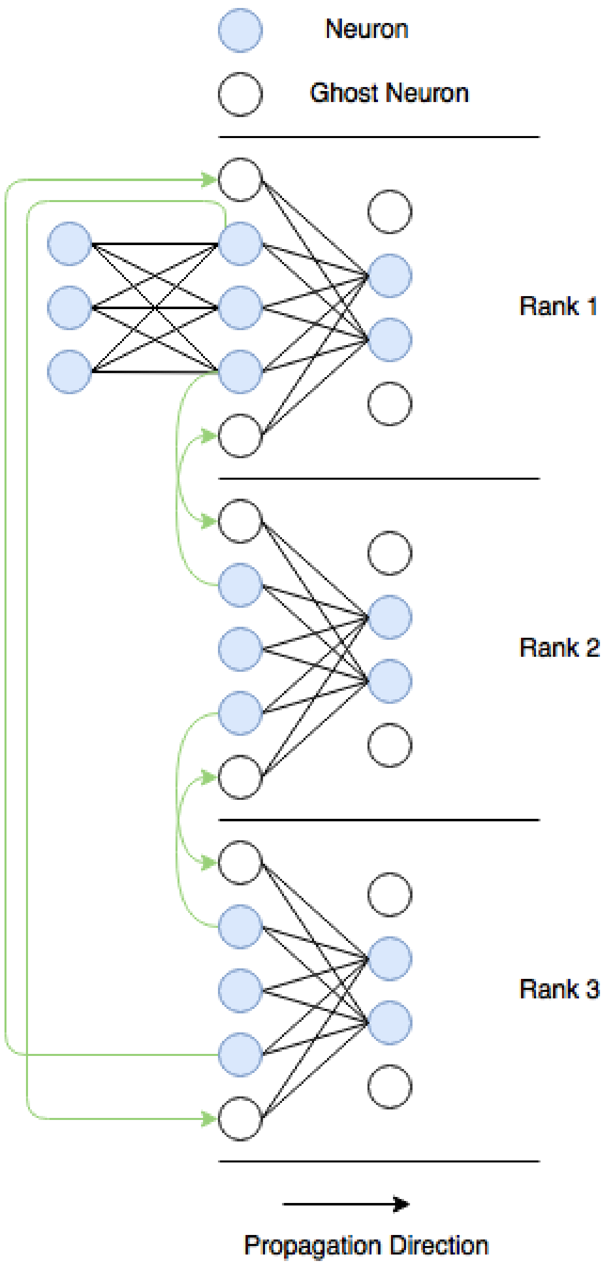

Fig. 3. Parallel Model Representation

Algorithm 1 Parallel Forward Propagation for each layer

```
1: $nr$ = number of ranks
2: $mr$ = rank of current process (myRank)
3: $fno(l)$ = first neuron output of layer $l$
4: $lno(l)$ = last neuron output of layer $l$
5: $R_r(l, gb)$ = output value of bottom ghost neuron in rank $r$ in layer $l$
6: $R_r(l, gt)$ = output value of top ghost neuron in rank $r$ in layer $l$
7:
8: if $myRank \geq 0$ then
9:     $net \leftarrow$ feedInputValuesToFirstLayer
10:    for each layer $l \in \mathcal{L}$ do
11:        $prevLayerNeurons \leftarrow$ getPrevLayerNrns($l$)
12:        for each neuron $n \in l(neurons)$ do
13:            $n \leftarrow$ feedForward(prevLayerNeurons)
14:        end for
15:        if $l \neq$ output then
16:            if $myRank = 1$ then
17:                $send(fno(l)) \rightarrow R_{nr-1}(l, gb)$
18:                $send(lno(l)) \rightarrow R_2(l, gt)$
19:                $ghostBtm \leftarrow Recv(R_2(fno(l))))$
20:                $ghostTop \leftarrow Recv(R_{nr-1}(lno(l))))$
21:            else if $myRank = nr - 1$ then
22:                $send(fno(l)) \rightarrow R_{mr-1}(l, gb)$
23:                $send(lno(l)) \rightarrow R_1(l, gt)$
24:                $ghostBtm \leftarrow Recv(R_1(fno(l))))$
25:                $ghostTop \leftarrow Recv(R_{mr-1}(lno(l))))$
26:            else
27:                $send(fno(l)) \rightarrow R_{mr-1}(l, gb)$
28:                $send(lno(l)) \rightarrow R_{mr+1}(l, gt)$
29:                $ghostBtm \leftarrow Recv(R_{mr+1}(fno(l))))$
30:                $ghostTop \leftarrow Recv(R_{mr-1}(lno(l))))$
31:            end if
32:        end if
33:    end for
34:    $Barrier()$
35: end if
```

B. Backward Propagation

Before the backward propagation process can take place, we first compute the total error/loss produced by the network for the given input sample. Since each rank computes its own output value we use *MPI_Allgather()* to communicate each rank's individual contribution to the output to all other ranks, thus allowing each rank to know the complete output. When then compute the loss and the gradient of the output. Since each rank is essentially training small portion of the total network the backpropagation step is not any different than it would be if the program is run in serial. We use the equations described in this paper to calculate the gradient value for each neuron. We first begin by assigning the correct calculated output gradients to each output neuron. Then we then calculate the and assign the gradients to each of the neurons in the hidden layers. Finally, we update the weights according to the gradient descent rule.

V. RESULTS

In this section we will provide an overview and an analysis of our performance results. In order to performance test our model we ran in on Rensselaer Polytechnic Institute's AMOS supercomputer. AMOS runs on the IBM Blue Gene/Q architecture. Table 1 describes the three different model architecture we performance tested.

TABLE I
MODEL ARCHITECTURES TESTED

Model	Input	H1	H2	H3	Output	Weights
S	3	64	128	256	64	57536
M	2048	4096	4096	4096	2048	50331648
L	2048	32768	8192	4096	2048	377487360

TABLE II
EXECUTION TIME AS NUMBER OF RANKS INCREASES FOR SMALL MODEL

Nodes	Ranks	Time Per Iter S (sec)	Speedup
1	2	17.452	1.0
1	5	0.4905	35.5
1	9	0.0928	188.1
1	17	0.0651	268.07
1	33	0.0354	497.20

TABLE III
EXECUTION TIME AS NUMBER OF RANKS IS INCREASED

Nodes	Ranks	Time Per Iter M (sec)	Time Per Iter L (sec)
1	33	67.499	2107.96
2	65	22.276	593.228
4	129	7.167	172.071
8	257	2.52	55.43
16	513	0.802	16.97
32	1025	0.255	6.44

A. Execution Time Performance

In order to performance test our framework we performed a strong scaling test on each of the model architectures from table 1. We first performed a simple strong scaling test on the small model by keeping the number of compute nodes a constant and increasing the number of ranks. The results, shown in table 2, show that execution time required to complete each iteration consistently decreases. Since our model uses a master rank to keep track of the overall model error and other statistics, running the model with 2 ranks is the equivalent of running in sequentially as only one rank is performing the forward and backward propagation algorithms. We are able to achieve a 497x speedup relative to sequential execution time when running with 33 ranks consisting of 1 master rank and 32 working ranks.

Furthermore, the performance results from the larger models are equally impressive. We did not perform a sequential execution time test for the medium and large models simply because it was taking too long. Thus, we started our tests

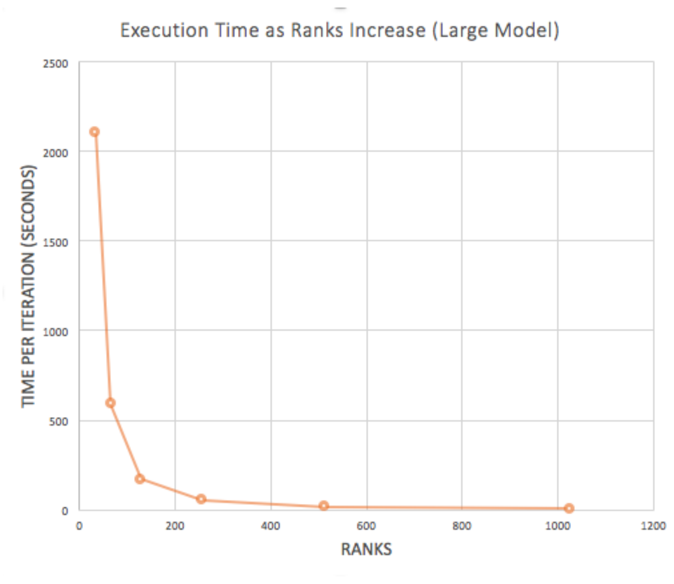

Fig. 4. Graph showing the decrease in execution time as the number of ranks is increased

with 33 ranks. Using 33 ranks the time required to perform on iteration on the large network was 2107 seconds. As we increased the number of ranks and nodes were able to reach a peak execution time of 6.4 seconds per iteration and max speedup of 327x using 32 compute nodes and 1025 ranks consisting of 1 master rank and 1024 working ranks.

TABLE IV
AMOUNT OF TIME SPENT ON COMMUNICATIONS BY THE LARGE MODEL

Nodes	Ranks	Max CPI (sec)	Avg CPI (sec)	% Time SC
1	33	21.415	7.930	1.015
2	65	6.305	1.398	1.062
4	129	2.371	0.634	1.378
8	257	1.047	0.238	1.890
16	513	0.482	0.100	2.844
32	1025	0.468	0.090	7.272

B. Performance Analysis

We conducted further performance analysis to determine the communication overhead of our framework. More specifically we determined the amount of time spent during the message passing step vs the time spent in actual computation. The resulting data, shown in table 4, contains the max communication time spent by the ranks per iteration (Max CPI), the average amount communication timer spent by the ranks per iteration (Avd CPI), and the percentage of the time spent by communication during each iteration. The results show that our framework is able to keep the communication overhead to a minimum. The general trend from the test results we collected shows that as we increase the number of ranks the percentage of time spent on communication increases. The reason behind this increase is due to the greater amount of interdependency that arises between ranks as the number of ranks is increased. As the number of ranks increases so does the number of *send()* and *recv()* operations. All receive operations must be satisfied for the forward propagation to move on to the next layer. In order for all the receives to be satisfied all the sends need to be successfully executed. This message passing process results an increased amount of interdependency as the number of ranks increases, which ultimately leads to a higher percentage of time being spent on communication as the number of ranks is increased.

VI. FUTURE WORK

The framework we have designed for this project is simply the beginning and we have chosen to continue further development on it in the future. One of the key aspects of the network that we would like to improve upon is accuracy. The primary motivation for this project was to reduce iteration execution time on very large scale networks we found. When testing our network we found it very difficult to find a dataset to which our test model (2048 input, 32768 hidden, 8192 hidden, 4096 hidden, and 2048 output) could fit to. Thus, for the purposes of testing our primary motive we generated dummy datasets to test how our parallelized framework improved iteration completion time. In the future we would like be able to either find or create a dataset that will allow our framework to learn even when running a massive model. It should be noted that we have confirmed that our framework is able to correctly to learn a smaller dataset when using a smaller more ”reasonable” model. Furthermore, in the future we would also like to redesign our parallel network structure to be fully connected. Our current structure is not fully connected because only the data from the rank boundary neurons is exchanged. We hypothesize that this will cause the network to learn more localized features and thus increase the likelihood over fitting to the training data when running larger models as the number of ranks is increased. In order to solve this issue we would like to redesign the network so that each neuron communicates its output value to each of the nodes in the next layer even if they are on different ranks using the *MPI_Bcast()* function. We originally did not try this method because we believe it will introduce significant overhead. However, if it allows the network to train more accurately it may be a model worth further exploring. There may be a sweet spot where a very large network is able to train sufficiently fast enough with better accuracy than our current model.

VII. CONCLUSION

In conclusion we were able to successfully achieve our goal of introducing parallelism to a neural network model in order to reduce the time needed for each forward and backward propagation iteration. More importantly this project served as a tremendous learning opportunity as we were able to learn about the inner workings of artificial neural networks by building a framework for one from scratch. Additionally, we also further strengthened and gain more knowledge on designing and building parallel applications and algorithms using OpenMPI.

VIII. TEAM MEMBER CONTRIBUTIONS

A. *Mukul Surajiwale*

Mukul designed the parallel architecture for the network and wrote a majority of the code base. More specifically he designed and implemented the object oriented class structure for the framework including the forward and backward propagation algorithms. Lastly, he also worked on conducting performance tests and writing the final paper.

B. *Rabiul Chowdhury*

Rabiul worked with Mukul to design the implement key portions of the network framework. He also took the initiative to add extensive error checking to in code to make sure potential problems are caught before they occur. He also aided in writing the final paper.

C. *Kyle Fawcett*

Kyle worked on writing the data preprocessing feature of the network. Additionally he also developed a script to convert image data into the format needed by our model. He also aided in contributing to the final paper.

D. *Alex Giris*

Alex aided in writing the final version of the paper. He ensured that it fully and accurately described our architecture.

IX. ADDITIONAL INFO

All the code for the project is located on Mukul's (PPC-surajm) Kratos account under the "Final Project" directory.

REFERENCES

[1] Bringing big data to the enterprise, IBM - What is big data?, 17-Mar-2017. [Online]. Available: http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html. [Accessed: 01-May-2017].

[2] Lyle N. Long and Ankur Gupta. "Scalable Massively Parallel Artificial Neural Networks", Journal of Aerospace Computing, Information, and Communication, Vol. 5, No. 1 (2008), pp. 3-15.

[3] The Increasing Value of Data, Future Agenda. [Online]. Available: http://www.futureagenda.org/insight/the-increasing-value-of-data. [Accessed: 01-May-2017].

[4] L. Mastin, Neurons & Synapses, The Human Memory, 2010. [Online]. Available: http://www.human-memory.net/brain_neurons.html. [Accessed: 01-May-2017].

[5] Exploring Deep Learning & CNNs, RSIP Vision, 03-Apr-2017. [Online]. Available: http://www.rsipvision.com/exploring-deep-learning/. [Accessed: 01-May-2017].

[6] Artificial Neural Networks Perceptron, https://en.wikibooks.org/wiki/Artificial-Neural-Networks/Print-Version/media/File:ArtificialNeuronModel-english.png

[7] G. M. Laskoski, "Addressing scaling and plasticity problems with a biologically motivated self-organizing network," 1990 IJCNN International Joint Conference on Neural Networks, San Diego, CA, USA, 1990, pp. 355-360 vol.2.

[8] G. Dahl, A. McAvinney, and T. Newhall, Parallelizing Neural Network Training For Cluster Systems, PDCN '08 Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks, pp. 220225, Feb. 12ADAD.