Parallelization of Feed-forward Fully Connected Neural Networks using CUDA and MPI

Mayur Ranchod
School of Computer Science and Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
1601745@students.wits.ac.za

Wesley Earl Stander
School of Computer Science and Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
1056114@students.wits.ac.za

***Abstract*—A neural network is a supervised machine-learning solution that is commonly used to perform a myriad of tasks which include speech recognition, image manipulation and classification. The widespread adoption of this solution follows from its simplicity and ability to achieve superior accuracy. Despite the attractiveness of this solution, it is evident that there is a significant computational cost involved in relation to the number of layers in the model, number of nodes in the model and the number of epochs used to train the model. In particular, we note that a handful of operations are repeatedly performed by all nodes in the network and the operations of nodes within a layer being independent of each other. These characteristics make a neural network a suitable candidate for parallelization and consequently provides the motivation for the report in which we investigate the performance gain achieved when implementing fully connected feed-forward neural networks on 2 parallel computing platforms, namely, MPI and CUDA. The performances achieved by our implemented models emphasize the effectiveness of parallelism as both models are able to achieve a noteworthy improvement in runtime performance.**

***Index Terms*—neural network, machine-learning, parallelization**

I. Introduction

Recent years have experienced a widespread adoption of machine-learning techniques to facilitate operations in a vast domain of fields. Machine-learning can be defined as the field of study of algorithms that enable systems to automatically learn and improve from experience without being explicitly programmed. These algorithms are primarily used to process large volumes of data to recognize patterns in the data and to gain insight which conduces well-informed decision making and accurate predictions. Despite the existence of numerous effective machine-learning solutions such as logistic regression and support vector machines to name a few, Artificial Neural Networks (ANNs) have been the dominant solution for performing a wide variety of tasks as a result its simplicity, versatility and efficacy which follows from its ability to model complex non-linear relationships. A brief introduction of the functioning of a neural network is provided below.

A neural network is a supervising learning solution used to perform a myriad of tasks such as classification and speech-recognition amongst many others and is designed in an attempt to mimic how the human brain functions. The human brain consists of millions of neurons which process information upon receiving a stimulus from another neuron. Upon receiving a stimulus, the neuron performs an operation and produces an output signal which is either accepted or rejected by another neuron which is dependent on the strength of the signal. To emulate the aforementioned process, a neural network consists of multiple nodes that are connected with an associated weight which indicates the strength of the connection. More specifically, the nodes represent the neurons while the connection weights are used to mimic the strength of the signal between neurons. The nodes are arranged in layers which form a hierarchy and also provides the network with the flexibility to adjust in response to the complexity of the task by varying the number of layers of nodes in the network. Each layer is characterized as either an input layer, hidden layer or an output layer.

Since a neural network is a supervised learning solution, the network receives a feature vector from a dataset as input. Upon receiving the feature vector, each node in the first layer computes the weighted sum between the elements in the feature vector and the weights between these elements and the node. To introduce non-linearity in the network and consequently to enable the network to model complex non-linear relationships, the weighted sum is passed to an activation function. Some of the commonly used activation functions include the hyperbolic tangent function, ReLU and the sigmoid/ logistic function as presented in (1). Once each node has computed its activation value, these values are used as the input to the following layer – networks that exhibit this pattern are referred to as feed-forward networks. This process is repeated until an output layer is reached which produces the output of the network. To introduce of the notion of learning which involves getting the network's output as close to the ground-truth as close as possible, a cost function such as cross-entropy loss in (2) is defined to quantify the error of the network's result which we wish to minimize. The process detailed above is referred to as the *forward propagation* of the network to produce an output.

As part of the training/ learning process, our objective is

to update the weights of the network such that the accuracy of the model's result increases as more training samples are considered. The most common approach to achieve this is by performing *backpropagation* which is an iterative process that starts with the last layer and moves backwards through the layers until the first layer is reached. The backpropagation algorithm relies on the gradient-descent algorithm in (3) to determine how the weights should be updated to minimize the cost function. As the model is trained on more samples, the model is able to produce more accurate results by means of numerous forward propagation and backpropagation operations.

In light of the training process described above, it is evident that the same operations are performed numerous times by nodes in the network i.e., each node computes its weighted sum and activation value in the forward pass and uses the errors received from next layer in the back pass. In particular, we notice that the operations performed by nodes within a layer are independent of each other. This repetitiveness and independence of the computations make the construction of a neural network a suitable candidate for parallelization. As a result, this observation provided the motivation for this report in which we attempted to implement a neural network using 2 different parallel computing platforms i.e., Message Passing Interface (MPI) and Compute Unified Device Architecture (CUDA). In particular, we considered a fully connected feed-forward neural network in which every node in the current layer is connected to every node in the next layer and the output of the current layer is the input to the next layer. The 2 parallel implementations were developed with the aim of reducing the computational runtime of the network by executing multiple independent operations simultaneously. We note that the approach taken by these 2 parallel frameworks differ to a great extent in the sense that MPI is a distributed memory framework whereas CUDA is a shared memory framework.

MPI operates by creating separate processes that executes the code and allows for communication between processes by passing messages between them. Both point-to-point and collective communication is supported and allows for a robust solution that is scalable, portable and high-performance. MPI is capable of executing on multiple nodes and having processes separated across nodes in a network allowing for usage on high performance clusters. In contrast, CUDA is designed to execute parallel algorithms on Nvidia Graphics Processing Units (GPU). Algorithms implemented using CUDA are able to reduce the execution time by using threads which execute segments of the algorithm concurrently. Both implementations are based on the C programming language.

We summarize the contribution of this report as follows:

- Implement a fully connected feed-forward neural network using CUDA.
- Implement a fully connected feed-forward neural network using MPI.
- Evaluate the performance gain achieved by the parallel implementations relative to a serial implementation.

This report is structured such that we present the methodology of our implementations in Section 2 which is followed by a description of the experimental setup presented in Section 3. Section 4 presents an analysis of results pertaining to our implementations. We conclude this report with Section 5 which highlights the salient points presented in this report.

II. Methodology

A. Serial Neural Network

1) Forward Propagation: As with all implementations that we have developed, we use the sigmoid function as the choice of activation function for all layers of the network. $\mathbf{x_i}$ represents the feature vector that is used as the input to the network. Thereafter, forward propagation is performed in a sequence in which the activation values of the current layer is used as input to the next layer. We define $\mathbf{x_0}$ as the bias term with value 1. $\mathbf{\Theta}$ is the parameter vector that is used to compute the weighted sum calculations for the nodes in the first hidden layer. The dimensionality of the feature vector is denoted by n.

$$\sigma(z) = \frac{1}{1+e^{-z}}, z = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n \quad (1)$$

The output of the network, denoted by $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)})$, is computed as the activation value of the node in the output layer. This value is within the range [0,1] and a threshold of 0.5 is used to perform a prediction. More specifically, if $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)}) > 0.5$, then the input sample is belongs to class 1 whereas $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)}) \leq 0.5$ indicates that the input sample belongs to class 0.

2) Backpropagation:

$$J(\mathbf{\Theta}) = -[y^{(n)} \log(h_{\mathbf{\Theta}}(\mathbf{x}^{(n)})) + (1-y^{(n)}) \log(1-h_{\mathbf{\Theta}}(\mathbf{x}^{(n)}))] \quad (2)$$

To train the network, backpropagation is performed to update the weights of the model such that the cost function $J(\mathbf{\Theta})$ (the cross-entropy error) is minimized, $y^{(n)}$ is the target label of the n^{th} datapoint while $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)})$ denotes the output produced by the model. Each θ_i is updated individually to minimize it's respective influence on the error of the model's prediction. Algorithm 1 presents the pseudocode which outlines the process to update the weights. N is the number of data-points for which learning is performed. All values of parameter vector $\mathbf{\Theta}$ are randomly initialized between 0 and 1. Each parameter θ is updated according to (3).

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial J(\mathbf{\Theta})}{\partial \theta_i} \quad (3)$$

Algorithm 1 Calculate weight parameters θ

```
epoch ← 1
while not converged do
  for n = 1 to N do
    compute h_Θ(x^(n))
    for θ_i in Θ do
      θ_i ← θ_i − α ∂J(Θ)/∂θ_i
    end for
    if is converged or epoch_max ≤ epoch then
      break
    end if
  end for
  epoch ← epoch + 1
end while
```

B. CUDA Implementation

Since this implementation makes extensive use of elaborately designed data structures, we begin by providing a brief discussion pertaining to their design. Since our network could potentially consist of multiple layers, we used a 3D tensor of matrices where each matrix contains the weights between 2 contiguous layers as illustrated in Fig. 1. All weights were randomly initialized between 0 and 1. For each matrix in the weight tensor, the number of rows was determined by the number of nodes in the current layer and the number of columns corresponded to the number of non-bias nodes in the next layer. The use of a 3D tensor significantly simplified the management of the network's weights and also allowed for an arbitrary number of nodes and layers that is user-specified. We note that all hidden layers contained the same number of nodes which could be user-specified. The other crucial data structure that we constructed was an *activation matrix* which was used to store the activation value (i.e., the result of passing the computed weighted sum to the sigmoid function) of each node. This design facilitated the forward propagation operation of the network as discussed in further detail below. Once the above data structures were defined, we commenced with the training process.

To easily train the model on every sample in the dataset, we utilized *Nvidia's dynamic parallelism* capabilities to ensure that new threads were spawned when a parent kernel invoked a child kernel. In particular, the parent kernel in our implementation was primarily responsible for extracting a sample from the dataset such that all threads operated on the current sample by invoking a child kernel to perform the forward and back propagation operations. Given a sample from the dataset, the contents were first copied to the first column of the activation matrix which enabled us to generalize the forward propagation process to an arbitrary number of layers by retrieving all activation values from a single source. The forward propagation operation was performed to compute the weighted sum of each node in the given

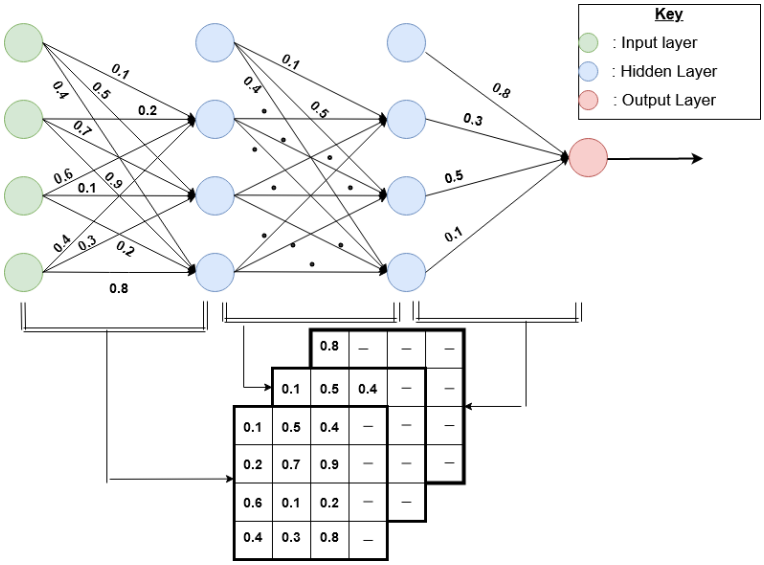

Fig. 1. Illustration of how the weights of the network were easily managed using a 3D tensor.

layer by performing matrix-vector multiplication between the previous layer's activation values and the corresponding weight matrix. We elucidate that the weight matrices in the 3D tensor were designed to be square in shape to easily perform the matrix vector multiplication and for all weight matrices to have the same dimensions. Since all the nodes in the current layer require the activation values from the previous layer, these values were loaded into shared memory to improve performance. Our automated process of forward propagation is summarised in the Algorithm 2 for brevity.

As a result of the inherent dependence of the current layer on the activation values of the previous layer, each thread was responsible for computing the weighted sum of a single node in the current layer. We expected the performance gain of the parallel implementation to increase as the number of layers and nodes increases. The forward propagation between consecutive layers was automated by the above process with the exception of the nodes in the output layer. This followed from the fact that the output layer typically contains fewer nodes than those contained in other layers of the network. In particular, we noted that the design of our implementation was easily amendable to account for an arbitrary number of nodes in the output layer. Since our implemented network was designed to perform binary classification on the chosen dataset presented in Section 3, it sufficed to use a single node in the output layer for which we performed a weighted reduction operation between the activation values of the preceding layer and the final weight matrix to produce the network's output.

To simplify the design of the implementation, we designed a kernel that was solely responsible for performing the reduction operation which utilized shared memory in an attempt to

Algorithm 2 Automated Forward Propagation

Define: $layers$: Number of layers
$layer$: The current layer
$tindex$: The ID of the current thread
$cache$: Shared memory
$input\_size$: Dimensions of the feature vector
act_mat: The activation matrix
$target$: The target vector of the dataset
f_vect: Feature vectors
w_tensor: Weight tensor

Invoking Kernel
entryNN ($f\_vect, w\_tensor, act\_mat, target$)*:*
for $n = 1$ to $layers - 1$ **do**
  MatVectMult($w\_tensor, act\_mat, n - 1$)
end for

Invoked Kernel
MatVectMult($w\_tensor, act\_mat, layer$)*:*
for $i = 0$ to $input\_size$ **do**
  act_mat[tindex+1][layer+1]+=cache[i]×w_tensor[i][tindex]
end for
act_mat[tindex+1][layer+1]=σ(act_mat[tindex+1][layer+1])

improve performance. Since this kernel was invoked by the current child kernel, this resulted in threads returning from the reduction kernel asynchronously which proved to be an issue that caused *race conditions* to occur when larger networks were used. This issue was exacerbated by the fact that synchronization could not be performed using the standard *__syncthreads()* function as this function performs synchronization only at a block scope. To remediate this issue, we first considered to perform the reduction operation in a serial manner under the premise that a sufficient speedup has been achieved by the forward propagation in the previous layers and noted that this serial operation would only be performed once for each sample. The other approach that was considered followed from the fact that the implementation utilized *Nvidia's dynamic parallelism* functionality which in turn, enabled us to use the *cudaDeviceSynchronize()* function that synchronizes the entire grid of threads. We acknowledged that both solutions impeded the performance gain achieved to a minor extent whilst experimentation revealed that the serial approach was able to achieve a better performance. Since our implementation contained a single node in the output layer, it was an understandable choice to designate a single thread to compute the activation value of the output layer and to compute the error of the network's output using (2). We note that this procedure would vary relative to the number of nodes in the output layer. All operations up to this point constituted of the forward pass of the network.

To update the weights of the network to produce more accurate predictions, we performed *backpropagation* which uses the evaluation of the network's cost function as an indication of how the weights should be adjusted. To simplify the process of backpropagation, we repurposed the activation matrix such that the value corresponding to node $u_i^{(j)}$ was overwritten to store $\frac{\partial J(\Theta)}{\partial u_i^{(j)}} \frac{\partial u_i^{(j)}}{\partial s_i^{(j)}}$. The motivation for this operation is made clearer upon considering the terms required to update a weight w_i originating from an arbitrary node $u_i^{(j-1)}$ in layer $j-1$ and leading to node $u_i^{(j)}$ in layer j. We see that the corresponding weight update would be expressed as:

$$\frac{\partial J(\mathbf{\Theta})}{\partial w_i} = \frac{\partial J(\Theta)}{\partial u_i^{(j)}} \frac{\partial u_i^{(j)}}{\partial s_i^{(j)}} \frac{\partial s_i^{(j)}}{\partial w_i} \tag{4}$$

The choice of the new value stored in the activation matrix followed from the observation that every weight leading to node $u_i^{(j)}$ involved an update rule that consisted of using $\frac{\partial J(\Theta)}{\partial u_i^{(j)}}$ and $\frac{\partial u_i^{(j)}}{\partial s_i^{(j)}}$ jointly. The weight update was completed by performing stochastic gradient descent (as presented in (3)) and multiplying the newly stored value by the activation of w_i's source node. By following this approach, we noticed that updating the weights between 2 consecutive layers only required the stored value of the nodes in the destination layer which obviated the need to traverse the network to compute the partial derivative of the cost function with respect to the destination node for every weight update. To perform backpropagation, each thread was assigned a node in the current layer with the responsibility of performing all weight updates that originated from its node. Upon completing the weight updates between the current layer and the preceding layer, we computed the stored value for the nodes in the preceding layer which would be used to update the weights that lead to that layer. This process is repeatedly performed in a function which we outline briefly in the pseudocode below.

Algorithm 3 Automated Weight Update Function

The definitions used below are presented in Algorithm 2.
old_activation= act_matrix[tindex][layer-1]
for $i = 0$ to $input\_size - 2$ **do**
  w_tensor[0][tindex][i]= w_tensor[layer][tindex][i]
  w_tensor[layer][tindex][i]=w_tensor[layer][tindex][i]-α×act_mat[i+1][layer]×old_activation
end for
sum = 0
for $i = 0$ to $input\_size - 2$ **do**
  sum += act_mat[i+1][layer] × w_tensor[0][tindex][i]
end for
act_mat[tindex][layer-1]=sum×old_activation×(1-old_activation)

Upon reviewing the above pseudocode, we observe that we first perform the weight updates between the current layer and the preceding layer which is then followed by computing the updated stored value of the nodes in the preceding layer. As a result, this design allowed us to automate the backpropagation

process for an arbitrary number of layers in the network. The backpropagation process was terminated upon updating the weights between the first hidden layer and the input layer which allowed us to consider the next training sample which undergoes the same process. We present the results achieved by this implementation in Section 4.

C. MPI Implementation

Two approaches were attempted for the MPI implementation. The first approach that was attempted utilized a process for each node of the neural network and as the feature set increased the amount of processes required increased proportionally. The second approach divided the work of the serial approach by the amount of processes allocated and exploits node parallelism. The first and last layers are implemented then a general middle layer is implemented that can work for any amount of middle layers. The MPI neural network algorithms follow Algorithm 1 where the forward propagation is the step in which $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)})$ is computed and the back propagation is the step where all θ_i are updated. The weights are initialized analogous to the CUDA implementation.

The weights of the network were managed similar to Fig.1 with the exception that the array was flattened into a one dimensional array and the weights for the layers before the last layer are shifted one index to the right. The array held space to store the weights to the next layer bias node but were not used in any calculations. This caused an extra check at the calculations that did not use the first weight in the dot product. The first approach has the weights divided so that each node has only its respective weights to ensure that indexing is one dimensional. The second approach has the full theta array on each node and is synchronized after the updates so the indexing is two dimensional. The SIZE value corresponds to the amount of nodes including the bias node for each layer. The implementation for MPI assumes the same amount of nodes for each layer of size SIZE and a single node for the output layer. The input layer was excluded when specifying the number of layers in the network.

1) Approach One - Forward Propagation: The Θ values are scattered to their respective node and the respective feature value corresponding to the node. The first layer calculation occurs for each node where each node calculates one element of the dot product between $\mathbf{\Theta}$ and $\mathbf{x_i}$ for the activation function. Before each proceeding layer the partial value for each activation function needs to be gathered from all the previous nodes. The activation functions are done as required and the result for the bias node is the bias. The middle layers generate partial values similarly to the first layer but use the result of the previous layer instead of the feature vector. The generation of the partial value for all layers after the first layer is represented in Algorithm 4. The final layer reduces the previous layer of results to produce the final result after using the activation function on the final result. The final result is stored on all nodes. The full approach is described in Algorithm 8 in the Appendix section. The value in the square brackets correlates to the layer being worked on. Values with 1 subscript correlate to the node index that they go to and the direction of the forward propagation is from the first layer to output $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)})$.

Algorithm 4 Approach One - Partial Value Calculation

```
MPI_Alltoall of partial
result ← 0
if world rank != 0 then
  for i ← 0 to SIZE do
    result ← result + partial_i
  end for
  result ← σ(result)
else
  result ← bias
end if
for i = 1 to SIZE do
  partial_i ← θ_i[layerCount] × feature
end for
layerCount ← layerCount + 1
```

2) Approach One - Back Propagation: The error is calculated on each node as they all have the final result. The back propagation operation requires the gathering of the results from each layer to be used in the preceding layer. Before each layer, the results from the proceeding layer are gathered. A back propagation value is required for each non-bias node which is θ_i for the final layer that allows for calculations to be made for nodes before the last layer. This back value needs to be gathered for all nodes so that they can be used to calculate 2 layers back from the last layer. Each θ_i value is updated according to (3). The generation of $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_i}$ for each θ is described in Algorithm 5. The full back propagation is described in Algorithm 9 in the Appendix section (subsection C). The value in square brackets correlates to the current layer. Values with 1 subscript correlate to the node index that they go to and values with 2 subscripts correspond to the node they go to and the node they come from respectively. The direction of the back propagation is from the output $h_{\mathbf{\Theta}}(\mathbf{x}^{(n)})$ to the first layer.

3) Approach Two - Forward Propagation: The feature values, θ values and the target values are broadcasted from the first process to all other processes before the algorithm begins. The approach divides the serial implementation between the allocated processes. The loop to work out each activation function of the next layer is divided among the processes. After each layer is calculated, the values are broadcasted to the other processes from the process that calculated the value. The broadcast process requires a check to see if the last layer was fully utilized as per the division of work load. The middle layers utilize similar code as the first layer but

Algorithm 5 Approach One - θ update for each node

MPI_Allgather of $result[count-1]$
for $w \leftarrow 1$ to SIZE **do**
  $b \leftarrow 0$
  if $count+1 = layers-1$ **then**
    $b \leftarrow back_{0,w}$
  else
    $b \leftarrow \sum_{j=1}^{SIZE} back_{j,w}[count]$
  end if
  $last \leftarrow \frac{\partial \sigma}{\partial z}(result_w[count])$
  $back_{w,0}[count] \leftarrow \frac{\partial \sigma}{\partial z}(result_w[count]) \times b \times \theta_w[count]$
  MPI_Allgather of $back_w[count]$
  update $\theta_w[count]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_w} = last \times b \times error \times result_{worldrank}[count-1]$
end for
$count \leftarrow count - 1$

calculate the results of the activation function using the results of the last layer and is described in Algorithm 6. The final result is calculated partially by each node and reduced. The full algorithm is represented in Algorithm 10 in the Appendix section. The indexing is the same as in the forward propagation in approach one.

Algorithm 6 Approach Two - calculate results of each node

$subSize \leftarrow \lceil \text{SIZE}/world\_size \rceil$
for $i \leftarrow subSize \times (world\_rank)$ to $subSize \times (world\_rank+1)$ and $i <$ SIZE **do**
  $result_i[layerCount] \leftarrow 0$
  if i = 0 **then**
    $result_i[layerCount] \leftarrow$ bias
  else
    $result_i[layerCount] \leftarrow \sigma(\sum_{j=0}^{SIZE} \theta_{j,i} \times result_j[layerCount-1])$
  end if
end for
Synchronize $result$ with MPI_Bcast
$layerCount \leftarrow layerCount + 1$

4) Approach Two - Back Propagation: The error is calculated on each node as they all have the final result. The loop to update each θ_i value for each layer is divided among the processes allocated. Before each layer the results from the proceeding layer are synchronized with a broadcast. A back propagation value is required for each non-bias node which is θ_i for the final layer that allows for calculations to be made for nodes before the last layer. This back value needs to be broadcasted for all nodes from the node that calculated it so that they can use them to calculate 2 layers back from the last layer. Each theta value is updated according to equation 3. The generation of $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_i}$ for each θ as well as the back propagation is described in Algorithm 7. The full algorithm is described in Algorithm 11 in Appendix C. The indexing is the same as the indexing in the back propagation of approach one.

Algorithm 7 Approach Two - θ Update

$backSize \leftarrow \lceil \text{SIZE} \times \text{SIZE}/world\_size \rceil$
for $i \leftarrow backSize \times (world\_rank)$ to $backSize \times (world\_rank+1)$ and $i < backSize \times world\_size$ **do**
  if i%SIZE> 0 **then**
    $b \leftarrow 0$
    if $count+1 = layers$ **then**
      $b \leftarrow back_{i\%SIZE,0}[count-1]$
    else
      $b \leftarrow \sum_{j=1}^{\text{SIZE}} back_{i,j}[count-1]$
    end if
    $last \leftarrow \frac{\partial \sigma}{\partial z}(result_{i\%SIZE}[count])$
    $back_{i/SIZE,i\%SIZE}[count-2] \leftarrow \frac{\partial \sigma}{\partial z}(result_{i\%SIZE}[count]) \times b \times theta_{i/SIZE,i\%SIZE}[count]$
    update $\theta_i[count]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_i} = last \times b \times error \times result_{worldrank}[count-1]$
  end if
end for
synchronize θ and $back$ with MPI_Bcast
$count \leftarrow count - 1$

III. Experimental Setup

A. Data Description

Since a neural network is a supervised learning solution, this indicates that a labelled dataset is required to train the model. For our implementation, we used the Heart Disease UCI dataset [1] which contains the health records of 303 patients which details the following factors:

- Age (in years)
- Sex (value 0: Female; value 1: Male)
- Chest pain type (value 1: typical type 1 angina; value 2: typical type angina; value 3: non-angina pain; value 4: asymptomatic)
- Resting blood pressure (measured in mm/Hg)
- Serum cholesterol (measured in mg/dl)
- Fasting blood sugar (value 0: <120 mg/dl; value 1: >120 mg/dl)
- Resting electrocardiographic results (value 0: normal; value 1: having ST-T wave abnormality; value 2: showing probable or definite left ventricular hypertrophy
- Maximum heart rate achieved
- Exercise-induced angina (value 0: No; value 1: Yes)
- Oldpeak- ST depression induced by exercise relative to rest
- Slope: the slope of the peak exercise ST segment (value 1: unsloping; value 2: flat; value 3: downsloping)
- Number of major vessels coloured by fluoroscopy (value 0-3)
- The result of performing a Thallium stress test (value 3: normal; value 6: fixed defect; value 7: reversible defect)

Using the above factors, we trained our model to perform binary classification which determines whether the patient has a heart disease or not (value 0: the patient has a heart disease;

value 1: the patient does not have a heart disease). As part of pre-processing the data, the data was split such that 80% of the data (243 samples) was used to train the model while 20% of the data (60 samples) was used for testing. The testing data was balanced such that there were an equal number of positive and negative examples. To improve the accuracy of the model, the data was normalized using *min-max normalization* which maps the minimum value of a feature to 0 and the maximum value to 1. This was achieved by determining the minimum of each feature and computing (5) for each value that belongs to that feature. Before using the data to train the model, a bias value equal to 1 was appended to each feature vector which allowed for the activation function to be shifted to the left or right to fit the data better.

$$value = \frac{value - min}{max - min} \tag{5}$$

B. Hardware Description

In order to determine the performance achieved by the aforementioned implementations, an array of hardware was utilized. More specifically, the MPI implementation was executed on a cluster with the following specifications:

- 100 nodes
- Each node has an i7 (with 4 cores), and 4GB RAM
- Each node has a GPU 750ti (640 cores, 2GB RAM)
- Connected by Ethernet
- OS: Ubuntu

To execute the MPI implementations on a local machine, a computer with the following specifications was used to conduct the experiments presented in Section 4. **Note: an excerpt of evidence of utilizing the cluster to conduct experiments is provided in the Appendix section.**

- CPU 9th Generation Intel® Core™ i7-9750H 6 Cores - 12 Threads
- RAM 64 GB DDR4 2666 MHz
- 2 TB PCIe SSD
- OS: Ubuntu

Since CUDA is a parallel computing framework that executes programs on the GPU, we present the specifications of the Nvidia GPU used to conduct the experiments below.

- CUDA Driver Version / Runtime Version 10.2 / 10.2
- Name: "GeForce GTX 1650"
- CUDA Capability Major/Minor version number: 7.5
- — Memory information for device —
- Total global mem: 4096 MB
- Total constant mem: 65536 B
- The size of shared memory per block: 49152 B
- The maximum number of registers per block: 65536
- The number of SMs on the device: 16
- The number of threads in a warp: 32
- The maximal number of threads allowed in a block: 1024
- Max thread dimensions (x,y,z): (1024, 1024, 64)
- Max grid dimensions (x,y,z): (2147483647, 65535, 65535)

C. Performance Metrics

In order to measure the performance gain achieved by the developed implementations, we computed the following performance metrics:

Speedup – The ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with p processing elements. The formula used is as follows:

$$S = T_s/T_p$$

where T_s is the serial time and T_p is the parallel time.

Throughput - The number of computing tasks performed per time unit (which we measure in Giga Floating-Point Operations per second (GFLOP/s)) and is determined using the following formula:

$$\text{Throughput=(number of operations) / (execution time in seconds} \times 10^9\text{)}$$

The calculation used to compute the number of operations is presented in the Appendix section (subsection A).

IV. Evaluation Results and Discussions

Since the primary objective of machine-learning models (such as neural networks) is to perform accurate predictions, it is crucial that these models are able to achieve good performance whilst ensuring that the correctness of these models are maintained. Following from this premise, the correctness of both presented implementations were verified by measuring the accuracy achieved on the testing dataset. Experimentation revealed that both models were able to achieve a promising accuracy of 86%. This was the highest accuracy achieved by both models and was achieved when the models were trained for 1000 epochs and 3 layers. Since the accuracies achieved by both models coincided with each other, we present the testing accuracy only in the MPI implementation to avoid the redundancy of computing the same computation unnecessarily. Upon reviewing the accuracies achieved when varying the number of layer, we noticed that networks consisting of fewer layers were able to achieve a better accuracy than models that consisted of many more layers. We attributed this to the fact that models with many layers were easily susceptible to overfitting which resulted in their poor performance. We elucidate that the focus of this report is rather concerned with the performance gains achieved as opposed to the accuracy achieved, consequently, a detailed analysis with regard to the performances achieved is presented below.

Fig. 2 presents the speedup achieved by the models when trained for 5 epochs while the number of layers were varied which is followed by Fig. 3. which illustrates the speedup achieved by the models consisting of 5 layers when the

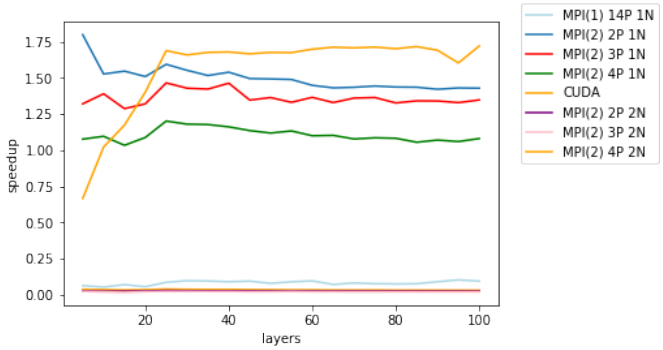

Fig. 2. Illustration of the speedup achieved when the models were trained for 5 epochs and the number of layers were varied.

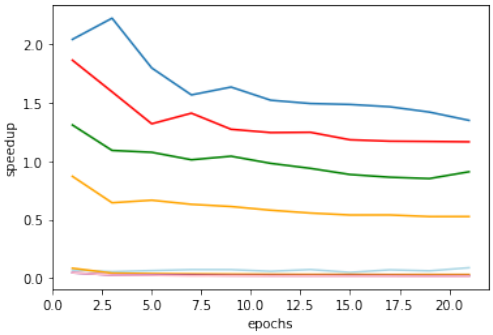

Fig. 3. Illustration of the speedup achieved by the models consisting of 5 layers while the number of epochs were varied.

number of epochs were varied. Upon reviewing Fig. 2, we see that the CUDA implementation achieved the greatest speedup of approximately 1.75x. In particular, we notice that a steady increase in speedup is achieved for models consisting of 3-20 layers and thereafter converges to achieve a speedup of 1.75x for models that consist of more than 20 layers. This speedup can be proven to be particularly advantageous when training models for real-world applications since the reduction in execution-time is considerable. An analysis of the CUDA implementation reveals that an MPI implementation is beneficial when training a model that consists of less than 20 layers since a greater speedup is achieved. Upon reviewing the results of the MPI implementations, we note that the legend corresponds to the implementation of MPI in parenthesis, the number of processes before P and the number of nodes before N. The experiments run on the cluster are the lines that have 2N to indicate that they were run across 2 nodes. The experiments that do not have a node count or are 1N were executed on a local machine. The performance achieved by approach one of the MPI implementation is considerably poor with execution times 20 times slower than the serial approach as presented in Fig. 2 and Fig. 3, therefore, the approach was not run on the cluster and was not used to measure the throughput. The lack of performance in the first approach can be attributed to the large amount of communication that occurs after each layer. Every node has to send and receive to every node at every layer and this caused huge performance losses. It can be seen that the trend in Fig.3 that the more processes allocated, the less the speedup for approach two of the MPI implementation. The lack of performance in approach one can also be attributed to the large amount of processes as a process was required for each node.

The first approach implemented for MPI suffered a flaw in design that required large amounts of communication alongside the large amount of processes. As a result, this approach achieved the worst performance as illustrated in Fig. 2 and Fig. 3. Due to the low performance on a single node, the algorithm was not executed on the cluster as its performance would be many times worse. This approach is not recommended to be the approach implemented for any neural network in MPI due to the poor performance and lack of scalability. The approach suffers as the data-set features are proportional to the amount of processes required and with data-sets with larger amounts of features, the amount of processes would cause far too much communication overhead and the processing would require virtual threads on machines that have less cores than required. The second approach performed significantly slower on two nodes as compared to one node. This disparity in performance is due to the communication calls between each layer being slowed by the ethernet communication line of the cluster. For the second MPI approach to be feasible on a cluster it will require a significantly faster means of communication between nodes such as fibre.

The second MPI approach performed poorly as the processes increased since the neural network node size is small at fourteen nodes. The communication operations in the algorithm rely on utilizing binary doubling and recursion in the communication of the MPI broadcast. This means that with a small network size the communication is similar to that of direct send and receives. This is inline with the performance of the node parallelism in [2]. If the network size was larger for a more complex data-set, the communication overhead would be significantly less and the speedup would increases with the amount of processes. Due to the large communication overhead and the proportional increases in communication when processes are increased, the performance suffers greatly as each process is added. It can be seen in Fig. 4 that the performance losses from three to four processes is greater than the performance losses from two to three processes due to the communication costs increasing proportionally to the processes.

Both implementations exhibit a similar trend when the number of epochs are varied as represented in Fig. 3. This is attributed to the fact that all implementations do not utilize techniques to optimize along the epochs and treat each epoch as a consecutive iteration of the algorithm. The trend indicates that an increase in epochs reduces the speedup as the

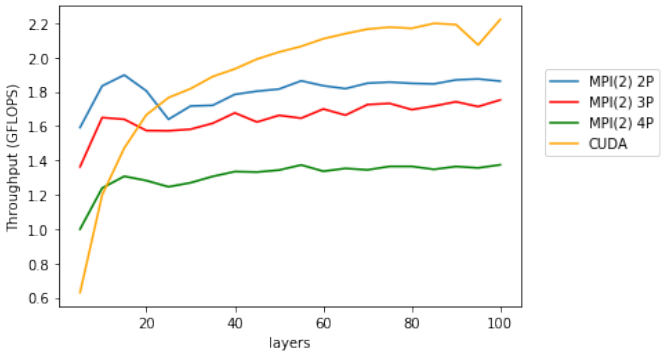

Fig. 4. Throughput achieved when the models were trained for 5 epochs and the number of layers were varied.

implementation of the MPI approach two tends towards the serial implementation at a speedup of 1x. Upon considering the results achieved by the CUDA implementation, we notice that this implementation achieved the greatest speedup when the number of layers were increased and the number of epochs were kept constant, however, this result differs considerably to Fig. 3 since that this approach is now inferior to the MPI implementations. We attribute this discrepancy due to the method in which the reduction operation (between the last hidden layer and the output layer) is performed. We remind the reader that this operation was performed in serial to avoid race conditions from arising. As a result, we observe that the number of reduction operations is constant when the number of epochs is constant and the number of layers were increased. In contrast, when the number of layers are constant and the number of epochs are increased, the frequency of executing the reduction operation is proportional to the increase in epochs performed. This increase in the number of executions of the reduction operation accentuates the bottleneck produced by the serial segment of the algorithm as the number of epochs are increased. We can deduce that our CUDA implementation is not suitable when the model is trained for an excessive number of epochs and a low amount of layers.

Fig. 4 provides a closer analysis of the best performing MPI implementations and the CUDA implementation. It can be seen that at the lowest layers the MPI approach performed better with regard to throughput as opposed to the CUDA implementation when the model consists of less than 20 layers. Similarly, we see that models consisting of more than 20 layers achieve the greatest throughput when the CUDA implementation is executed. Upon evaluating the MPI implementation against the implementation in [3], it can be seen that performance increases with the amount of processes they allocate utilizing a Levenberg-Marquardt based algorithm. The performance of this implementation has the highest speeds at 2 processes and hence is not optimal compared to the aforementioned implementation. The implementation in [2] utilizes training example parallelism to achieve much higher performance. Their node parallelism technique achieves similar results and the linear mapping of nodes to processes is poor performing as in MPI approach one. The trend of node parallelism with smaller neural networks as presented in [2] indicates that with less than 500 nodes, more processes leads to a decrease in speedup and only after 500 nodes does the speedup increase with the increase of processes and this is inline with the results discovered in [2] for MPI approach two. The highest speedup achieved with node parallelism was around 2 with 100 nodes and 2 processes. Similar results were achieved in this paper with 13 nodes and 2 processes.

It can be observed from the graphs that the performance of MPI approach two performs worse as the processes is increased but performs similarly regardless of the layer size and the epoch size on each process respectively. This is due to the approach exploiting node parallelism and the speedup would be most as the number of nodes of the network increase. The number of layers and the number of epochs do not affect the performance of the MPI approaches as they do not exploit training parallelism or layer parallelism. In contrast, we see that the applicability of the CUDA implementation is highly dependent on the number of layers and is a promising approach in the event that the model consists of an excessive number of layers and trained for a modest number of epochs.

V. Conclusion

Taking the analysis presented in Section 4 into consideration, we see that the parallel implementations are capable of achieving significant speedup in comparison to a serial implementation.

References

[1] ronit, "Heart disease uci," 2018, https://www.kaggle.com/ronitf/heart-disease-uci, Accessed on: 20 June 2020.

[2] M. Pethick, M. Liddle, P. Werstein, and Z. Huang, "Parallelization of a backpropagation neural network on a cluster computer," in *International conference on parallel and distributed computing and systems (PDCS 2003)*, 2003.

[3] U. Lotrič and A. Dobnikar, "Parallel implementations of feed-forward neural network using mpi and c# on. net platform," in *Adaptive and Natural Computing Algorithms*. Springer, 2005, pp. 534–537.

APPENDIX

A. Computing the Number of operations performed

We present the calculation used to determine the number of operations performed in (A). We use the following definitions to simplify the expression.

- i: Dimensionality of the samples
- $layers$: The number of layers in the network
- $epochs$: Number of epochs performed
- $size$: number of samples

$$\begin{aligned} operations = [[(i-1) \times (i + (i-1) + 3)] \times (layers - 1) \\ +(i + (i-1) + 3) + i \times 4 + i \times (i-1) \times 8 \\ +[[(i-1) \times 6 + 6] \times i(i-1)] \times (layers - 2)] \times epochs \times size \end{aligned} \tag{6}$$

B. Evidence of cluster usage

Processes were distributed among 2 nodes.

Amount of Processes	layers	epochs	speedup
2	5	1	0.109427
2	5	3	0.039146
2	5	5	0.046220
2	5	7	0.042475
2	5	9	0.032373
2	5	11	0.034447
2	5	13	0.029031
2	5	15	0.029778
2	5	17	0.031754
2	5	19	0.032629
2	5	21	0.032184

C. MPI pseudo-code

Algorithm 8 Approach one - forward propagation

for $i \leftarrow 1$ to SIZE **do**
  $partial_i \leftarrow \theta_i[layerCount] \times feature$
end for
if layers > 2 **then**
  for $l \leftarrow 1$ to $layers$-1 **do**
    MPI_Alltoall of $partial$
    $result \leftarrow 0$
    if world rank != 0 **then**
      for $i \leftarrow 0$ to SIZE **do**
        $result \leftarrow result + partial_i$
      end for
      $result \leftarrow \sigma(result)$
    else
      $result \leftarrow$ bias
    end if
    for $i = 1$ to SIZE **do**
      $partial_i \leftarrow \theta_i[layerCount] \times feature$
    end for
    $layerCount$
  end for
end if
MPI_Alltoall of $partial$
$result \leftarrow 0$
if world rank != 0 **then**
  for $i \leftarrow 0$ to SIZE **do**
    $result \leftarrow result + partial_i$
  end for
  $result \leftarrow \sigma(result)$
else
  $result \leftarrow$ bias
end if
$partialResult \leftarrow result \times \theta_0[layerCount]$
MPI_Allreduce of $partialResult$
$finalResult \leftarrow \sigma(partialResult)$

Algorithm 9 Approach one - back propagation

$count \leftarrow layers - 2$
MPI_Allgather of $result[count - 1]$
$error \leftarrow h_{\mathbf{\Theta}}(x) - target$
$back_{0,0}[count] \leftarrow \theta_0[count + 1]$
MPI_Allgather of $back[count]$
update $\theta_0[count + 1]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_w} = error \times result_{worldrank}[count]$
if layers > 2 **then**
  for $l \leftarrow 1$ to $layers$-1 **do**
    MPI_Allgather of $result[count - 1]$
    for $w \leftarrow 1$ to SIZE **do**
      $b \leftarrow 0$
      if $count + 1 = layers - 1$ **then**
        $b \leftarrow back_{0,w}$
      else
        $b \leftarrow \sum_{j=1}^{SIZE} back_{j,w}[count]$
      end if
      $last \leftarrow \frac{\partial \sigma}{\partial z}(result_w[count])$
      $back_{w,0}[count] \leftarrow \frac{\partial \sigma}{\partial z}(result_w[count]) \times b \times theta_w[count]$
      MPI_Allgather of $back_w[count]$
      update $\theta_w[count]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_w} = last \times b \times error \times result_{worldrank}[count - 1]$
    end for
    $count \leftarrow count - 1$
  end for
end if
if $count + 1 = layers - 1$ **then**
  MPI_Allgather of $result[count]$
end if
for $w \leftarrow 1$ to SIZE **do**
  $b \leftarrow 0$
  if $count + 1 = layers - 1$ **then**
    $b \leftarrow back_{0,w}$
  else
    $b \leftarrow \sum_{j=1}^{SIZE} back_{j,w}[count]$
  end if
  $last \leftarrow \frac{\partial \sigma}{\partial z}(result_w[count])$
  update $\theta_w[count]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_w} = last \times b \times error \times feature$
end for

Algorithm 10 Approach two - forward propagation

$layerCount \leftarrow 0$
$subSize \leftarrow \lceil \text{SIZE}/world\_size \rceil$
for $i \leftarrow subSize \times (world\_rank)$ to $subSize \times (world\_rank + 1)$ and $i <$ SIZE **do**
  $result_i[layerCount] \leftarrow 0$
  if i = 0 **then**
    $result_i[layerCount] \leftarrow$ bias
  else
    $result_i[layerCount] \leftarrow \sigma(\sum_{j=0}^{SIZE} \theta_{j,i} \times feature_j)$
  end if
end for
Synchronize $result$ with MPI_Bcast
$layerCount \leftarrow layerCount + 1$
if layers > 2 **then**
  for $l \leftarrow 1$ to $layers$-1 **do**
    for $i \leftarrow subSize \times (world\_rank)$ to $subSize \times (world\_rank + 1)$ and $i <$ SIZE **do**
      $result_i[layerCount] \leftarrow 0$
      if i = 0 **then**
        $result_i[layerCount] \leftarrow$ bias
      else
        $result_i[layerCount] \leftarrow \sigma(\sum_{j=0}^{SIZE} \theta_{j,i} \times result_j[layerCount - 1])$
      end if
    end for
    Synchronize $result$ with MPI_Bcast
    $layerCount \leftarrow layerCount + 1$
  end for
end if
for $i \leftarrow subSize \times (world\_rank)$ to $subSize \times (world\_rank + 1)$ and $i <$ SIZE **do**
  $partialResult \leftarrow \theta_{0,i} \times result_i[layerCount - 1]$
end for
MPI_Allreduce of $partialResult$
$finalResult \leftarrow \sigma(partialResult)$

Algorithm 11 Approach two - back propagation

$count \leftarrow layers - 1$
$subSize \leftarrow \lceil \text{SIZE}/world\_size \rceil$
$backSize \leftarrow \lceil \text{SIZE} \times \text{SIZE}/world\_size \rceil$
$error \leftarrow h_{\mathbf{\Theta}}(x) - target$
for $i \leftarrow subSize \times (world\_rank)$ to $subSize \times (world\_rank + 1)$ and $i <$ SIZE **do**
  if i != 0 **then**
    $back_{i,0}[count - 1] \leftarrow \theta_0[count]$
  end if
  update $\theta_{i,0}[count + 1]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_w} = error \times result_{worldrank}[count]$
end for
synchronize θ and $back$ with MPI_Bcast
if layers > 2 **then**
  for each layer $l \leftarrow 1$ to $layers$-1 **do**
    for $i \leftarrow backSize \times (world\_rank)$ to $backSize \times (world\_rank + 1)$ and $i < backSize \times world\_size$ **do**
      if i%SIZE> 0 **then**
        $b \leftarrow 0$
        if $count + 1 = layers$ **then**
          $b \leftarrow back_{i\%SIZE,0}[count - 1]$
        else
          $b \leftarrow \sum_{j=1}^{\text{SIZE}} back_{i,j}[count - 1]$
        end if
        $last \leftarrow \frac{\partial \sigma}{\partial z}(result_{i\%SIZE}[count])$
        $back_{i/SIZE,i\%SIZE}[count - 2] \leftarrow \frac{\partial \sigma}{\partial z}(result_{i\%SIZE}[count]) \times b \times theta_{i/SIZE,i\%SIZE}[count]$
        update $\theta_i[count]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_i} = last \times b \times error \times result_{worldrank}[count - 1]$
      end if
    end for
    synchronize θ and $back$ with MPI_Bcast
    $count \leftarrow count - 1$
  end for
end if
for $i \leftarrow backSize \times (world\_rank)$ to $backSize \times (world\_rank + 1)$ and $i < backSize \times world\_size$ **do**
  if i%SIZE> 0 **then**
    $b \leftarrow 0$
    if $count + 1 = layers$ **then**
      $b \leftarrow back_{i\%SIZE,0}[count - 1]$
    else
      $b \leftarrow \sum_{j=1}^{\text{SIZE}} back_{i,j}[count - 1]$
    end if
    $last \leftarrow \frac{\partial \sigma}{\partial z}(result_{i\%SIZE}[count])$
    update $\theta_i[count]$ with $\frac{\partial J(\mathbf{\Theta})}{\partial \theta_i} = last \times b \times error \times feature_i$
  end if
end for
synchronize θ with MPI_Bcast