PARALLEL IMPLEMENTATION OF NEURAL NETWORKS

Anteneh Getachew
(FTP 0138/09)
School of Computer Engineering
Addis Ababa Science and Technology University
Addis Ababa, Ethiopia

Yishak Tadele
(FTP 1021/09)
School of Computer Engineering
Addis Ababa Science and Technology University
Addis Ababa, Ethiopia

ABSTRACT

Enormous computing power is required in today's generation where deep learning algorithms are applied in a lot of application domains. Deep learning and Neural network models work great when the amount of training data is huge, which increases the number of learnable parameters so have the time required to train them. Due to this reason, there needs an efficient computing method that parallelizes the training process. In this work, we parallelized the training of a two-layer neural network for hand-written digits recognition using the data and model parallelization techniques separately to boost the training time by taking the serial time as a benchmark in c programming using OpenMP and Pthread application programming interfaces (API). Findings suggest that we have achieved about 4.84 and 4 speedups using data and model parallelizations respectively. As a challenge, even though we got better speedups when we apply data parallelization we can not maintain the accuracy of the model as the number of threads increase.

***K*eywords** Deep Learning, Neural Network, Data parallelism, Model parallelism, Pthread, OpenMP

Introduction

Artificial Neural networks(ANNs) are relatively crude computerized models that imitate the human brain's neuronal structure, which contains three trillion neurons and they form a massively parallel information system[2]. These artificial neural networks make up the backbone of deep learning algorithms used in many applications today, ranging from image classification to autonomous cars. Deep Learning models trained on a large amount of data like VGG Face gave almost a perfect prediction accuracy that approaches the human ability to identify a human face[6]. But it takes almost a week to train that kind of model using a CPU machine.

Neural networks in general typically have millions of parameters and require large amounts of data to get the optimal value for these parameters of the model. With the growing size of the network and larger data-set, we can extract more complex representations, but at the cost of insanely high computation time, Some Neural Networks take weeks to train on a single-core CPU[3].

Deep learning models trained on large data sets have been widely successful in both vision and language domains. As state-of-the-art deep learning architectures have continued to grow in parameter count, so have the compute budgets and times required to train them, increasing the need for compute-efficient methods that parallelize training[5].

Two main approaches can be taken to parallelize the training of steps of a neural network, model parallelism and Data parallelism[7].

1. Model Parallelism assigns different sections of the network across multiple threads as shown in Figure 1. Each thread handles the node computation and it is responsible for and passes along the result to other threads which handle the node computation in the next neural network layer.
2. Data parallelism creates one network and instantiates copies of it across multiple threads as shown in Figure 3. This way each thread receives its network and a batch of data to train, and once all threads finish working on their batch, their network parameters are averaged with all other threads to produce one master network.

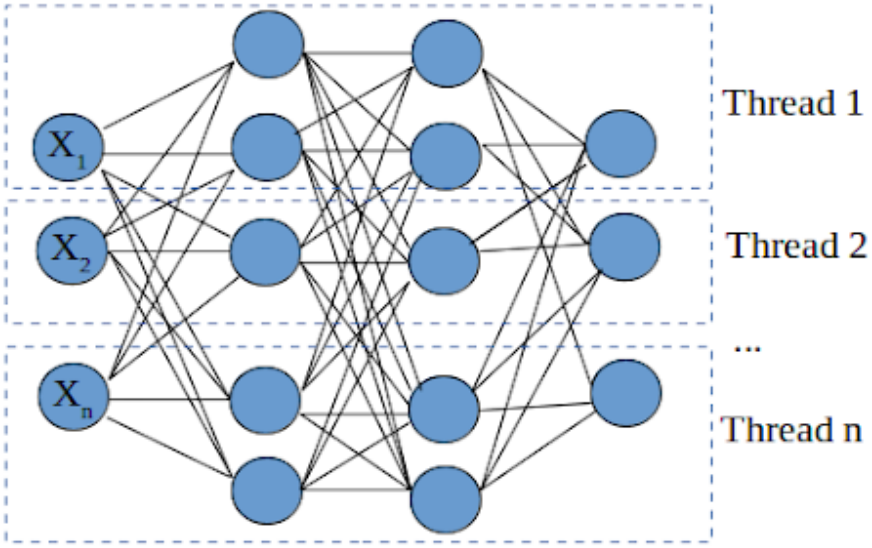

Figure 1: Model parallelism architecture

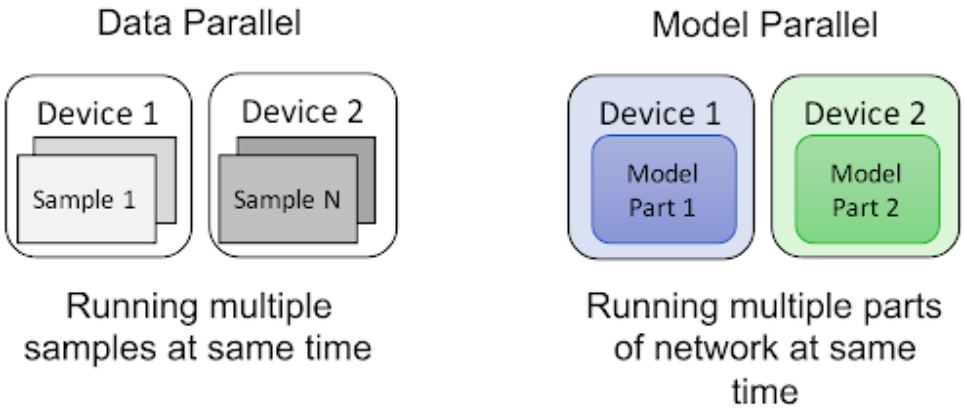

Figure 2: Parallel Neural Networks and Batch Sizes

In this work, we have implemented the parallel implementation of a two-layer neural network for handwritten digit recognition, both using data and model parallelism techniques. The rest of this paper is organized as follows. Section 2 provides Related works. Section 3 describes the Methodology we follow Section 4 provides the research gap and a critique summary. Section 5 and 6 provide results, discussion and conclusion for this survey.

Related Works

In our days multithreaded and multicore CPUs with shared memory are a cost-effective way of obtaining significant increases in CPU performance. An exponential growth in performance is expected in the near future from more hardware threads and cores per CPU[8].

In [7] they proposed a method for parallelization of neural network training based on the backpropagation algorithm and implemented it using two different multi-threading techniques (OpenMP and POSIX threads), and in case of the O2 A-Band spectrum simulation problem with almost two million patters they obtained an increase in computation speed very close to the expected ideal, that is 743% when 8 cores were used. The techniques MPI and OpenMP have been

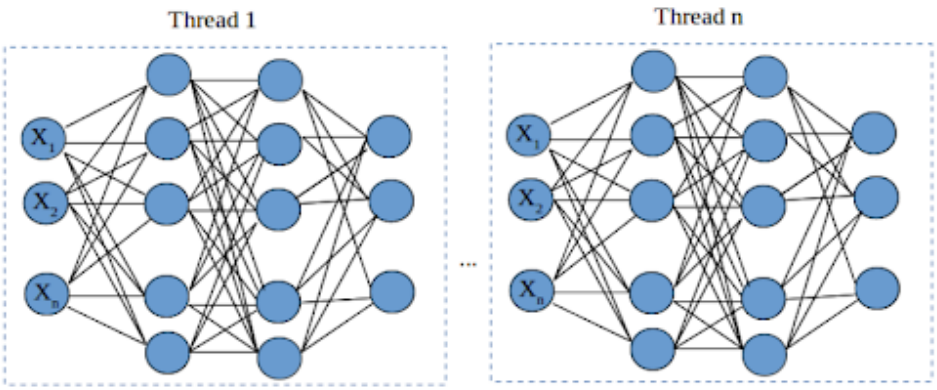

Figure 3: Data parallelism

applied to a number of applications for ANN parallel training, but the parallel execution of neurons at each level is highlighted very less[1].

Methodology

The serial code was found from [4] and the file read operation, which reads the image from the MNIST dataset list, has a high overhead which affected the training process. So we implemented another reading operation to read the file faster and store the result in a two-dimensional array. There are four variables created for this purpose. Train_image, Test_image, Train_label and Test_label. Train and test _image variables are 2D arrays that contain the image order number(60,000 images for train and 10,000 images for the test) and the image in 784(28x28) size.

Data Parallelism

The Data is divided for the thread number and the start and end image id is given for each thread to train the model with each image. Here we used Pthreads because they can be manipulated and assigned custom tasks easily. Each thread will have its own version of the model initialized and we created a new Structure called ThreadInfo, which consists of the starting and ending number of the image sequence and the initialized network as can be seen in Figure 4. Each thread is created and given this structure variable, so each will proceed with the model.

```
typedef struct threadInfo_{
  Network * network;
  int start, end;
}ThreadInfo;
```

Figure 4: C language Structure holding a Neural Network

In the beginning, the models' weight and back propagation value are null, so they are randomly initialized and given to the threads but after the first iteration, the weights are summed and averaged and broadcasted again for each thread. This is done for each node on each layer of the network.

Network Architecture

The network architecture that we have used in this work is a shallow two layer neural network to demonstrate how to make the neural network for efficient by parallelizing the jobs among multiple threads as compared to the serial version of it.The model architecture is shown in Figure 6.

```
 Serial Implementation:

        ========= Training Started ==========

Training epoch 1/10:    Test Accuracy: 93.58%
Training epoch 2/10:    Test Accuracy: 95.01%
Training epoch 3/10:    Test Accuracy: 95.41%
Training epoch 4/10:    Test Accuracy: 95.75%
Training epoch 5/10:    Test Accuracy: 95.94%
Training epoch 6/10:    Test Accuracy: 96.03%
Training epoch 7/10:    Test Accuracy: 96.13%
Training epoch 8/10:    Test Accuracy: 96.22%
Training epoch 9/10:    Test Accuracy: 96.21%
Training epoch 10/10:   Test Accuracy: 96.33%


        ========= Training Ended ==========


Time Elapsed=465.157187
```

Figure 5: Performance evaluation of Serial implementation

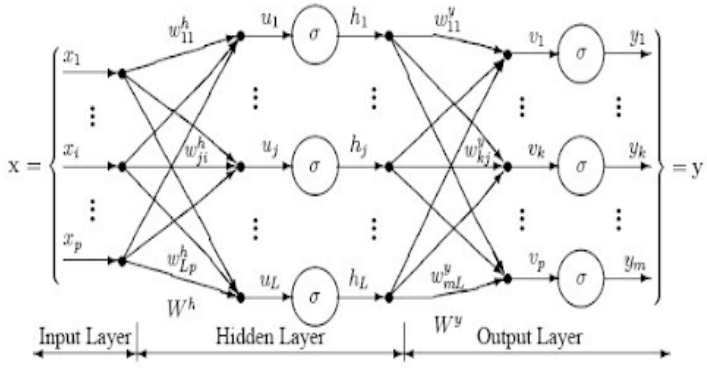

Figure 6: Neural Network Architecture

Evaluation metrics and result analysis

Our main evaluation metric is the speedup gained. By taking the serial version implementation as a benchmark we were able to observe the speedup increase obtained. Another metric is the accuracy gained at each iteration step. Accuracy is the measurement used to determine which model is best at identifying relationships and patterns between variables in a dataset based on the input, or training, data. The better a model can generalize to 'unseen' data, the better predictions and insights it can produce, which in turn deliver more business value. So this metric is affected by the amount of data the model is trained on, so even though increasing the number of threads speeds up the training, dividing the data to multiple threads will reduce the data that each model is trained on. This results in an accuracy decrease, as the number of threads is increased. It is shown in the following Figures 7 , and 8.

In this work we have found that as the number of worker threads increase, the speedup gain show an increasing trend until it saturates when the number of threads equal to the number of cores on the machine 12, as can be seen in the Figure 10. Although we get this achievement to the speed up as the number of threads increase , we can not maintain the accuracy of the model, which is very important for it to use in real world application . This challenge of not maintaining the accuracy of the model can be resolved by collecting huge amount of data which makes each worker thread to get enough data to learn from.

```
Number of threads: 1
Test Accuracy: 9.72%

        ========= Training Started ==========

Training epoch 1/10:    Test Accuracy: 93.86%
Training epoch 2/10:    Test Accuracy: 95.14%
Training epoch 3/10:    Test Accuracy: 95.79%
Training epoch 4/10:    Test Accuracy: 96.11%
Training epoch 5/10:    Test Accuracy: 96.35%
Training epoch 6/10:    Test Accuracy: 96.39%
Training epoch 7/10:    Test Accuracy: 96.57%
Training epoch 8/10:    Test Accuracy: 96.58%
Training epoch 9/10:    Test Accuracy: 96.46%
Training epoch 10/10:   Test Accuracy: 96.76%

        ========= Training Ended ==========

Parallel Implementation of NN

Time Elapsed=466.491147 seconds
```

```
Number of threads: 2
Test Accuracy: 7.53%

        ========= Training Started ==========

Training epoch 1/10:    Test Accuracy: 76.18%
Training epoch 2/10:    Test Accuracy: 92.28%
Training epoch 3/10:    Test Accuracy: 93.91%
Training epoch 4/10:    Test Accuracy: 94.69%
Training epoch 5/10:    Test Accuracy: 95.04%
Training epoch 6/10:    Test Accuracy: 95.50%
Training epoch 7/10:    Test Accuracy: 95.65%
Training epoch 8/10:    Test Accuracy: 95.69%
Training epoch 9/10:    Test Accuracy: 95.85%
Training epoch 10/10:   Test Accuracy: 95.81%

        ========= Training Ended ==========

Parallel Implementation of NN

Time Elapsed=258.826308 seconds
```

Figure 7: Performance of parallel implementation, numThreads = 1 , and numThreads = 2.

```
Number of threads: 3
Test Accuracy: 10.55%

        ========= Training Started ==========

Training epoch 1/10:    Test Accuracy: 70.64%
Training epoch 2/10:    Test Accuracy: 83.84%
Training epoch 3/10:    Test Accuracy: 85.64%
Training epoch 4/10:    Test Accuracy: 86.35%
Training epoch 5/10:    Test Accuracy: 86.34%
Training epoch 6/10:    Test Accuracy: 85.84%
Training epoch 7/10:    Test Accuracy: 85.42%
Training epoch 8/10:    Test Accuracy: 85.24%
Training epoch 9/10:    Test Accuracy: 85.08%
Training epoch 10/10:   Test Accuracy: 85.20%

        ========= Training Ended ==========

Parallel Implementation of NN

Time Elapsed=189.421886 seconds
```

```
Number of threads: 4
Test Accuracy: 11.44%

        ========= Training Started ==========

Training epoch 1/10:    Test Accuracy: 62.47%
Training epoch 2/10:    Test Accuracy: 81.76%
Training epoch 3/10:    Test Accuracy: 82.41%
Training epoch 4/10:    Test Accuracy: 80.36%
Training epoch 5/10:    Test Accuracy: 78.82%
Training epoch 6/10:    Test Accuracy: 77.74%
Training epoch 7/10:    Test Accuracy: 77.14%
Training epoch 8/10:    Test Accuracy: 76.35%
Training epoch 9/10:    Test Accuracy: 75.75%
Training epoch 10/10:   Test Accuracy: 75.48%

        ========= Training Ended ==========

Parallel Implementation of NN

Time Elapsed=150.738524 seconds
```

Figure 8: Performance of parallel implementation, numThreads = 3, and numThreads = 4.

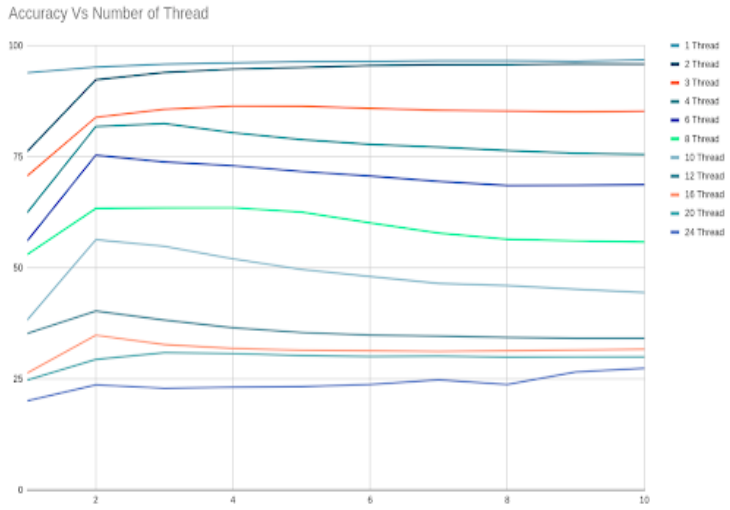

Figure 9: Accuracy Vs number of threads

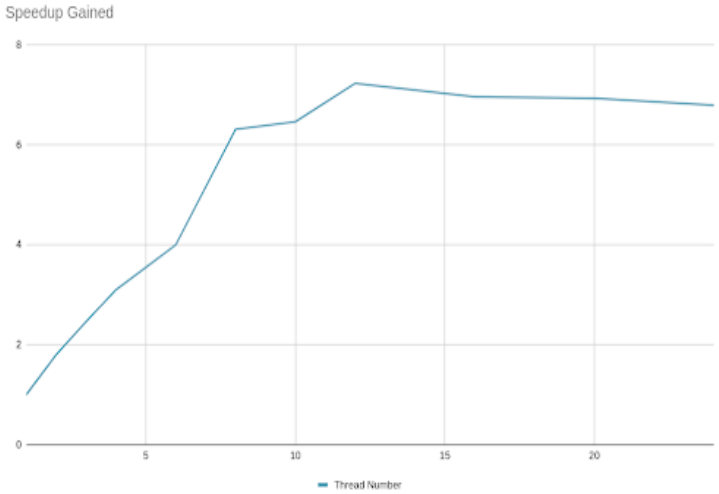

Figure 10: Speedup Gained Vs Thread Number

Discussion

As we can see from 9 as the number of the thread increases the accuracy decreases. This is because the amount of data distributed among each thread will reduce as the number of thread increases. From 10 we can see that the number of thread shows an increase on the speedup gain with the maximum value of 4.87 obtained at thread number 12. This outperforms the serial version.

Conclusion

Computing power is still a problem in a lot of machine learning as well as sophisticated simulations, so utilizing the processing power we have is not a choice, but must. Paralleling serial code is one of the approach to utilize the machine processing power in an efficient manner. In this work, we have tried to parallelize the training of a two-layer neural network for hand-written digits recognition using the data and model parallelization techniques separately to boost the training time by taking the serial time as a benchmark in c programming using OpenMP and Pthread application programming interfaces (API).Findings suggest that we have achieved about 4.84 and 4 speedups using data and model parallelizations respectively. As a challenge, even though we got better speedups when we apply data parallelization we can not maintain the accuracy of the model as the number of threads increase. As a future work we are planing to implement this model using GPU programming in CUDA and include regularization techniques to get rid of model over-fitting when the amount of training data is not that much huge.

References

[1] P. Chanthini and K. Shyamala. A survey on parallelization of neural network using MPI and open MP. 9.

[2] Wang Guoyin and Shi Hong-bao. Parallel neural network architectures. *Proceedings of ICNN'95 - International Conference on Neural Networks*, 3:1234–1239 vol.3, 1995.

[3] Vishakh Hegde and Sheema Usmani. Parallel and distributed deep learning. page 8.

[4] Dhawal Jain. dhawal777/handwritten-digit-recoganization-by-ANN-in-openmp. original-date: 2019-05-08T08:20:01Z.

[5] Michael Laskin, Luke Metz, Seth Nabarrao, Mark Saroufim, Badreddine Noune, Carlo Luschi, Jascha Sohl-Dickstein, and Pieter Abbeel. Parallel training of deep networks with local updates, 12 2020.

[6] Omkar Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. volume 1, pages 41.1–41.12, 01 2015.

[7] Olena Schuessler and Diego Loyola. Parallel training of artificial neural networks using multithreaded and multicore CPUs. In Andrej Dobnikar, Uroš Lotrič, and Branko Šter, editors, *Adaptive and Natural Computing Algorithms*, pages 70–79. Springer Berlin Heidelberg.

[8] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh. Parallelism via multithreaded and multicore CPUs. 3(43):24–32.