Brian Crafton
High Performance Computing
April 15, 2016

Parallelizing A Neural Network

Introduction:

Machine learning is a hot topic in the world of computer science and engineering. It makes use of statistics and data to recognize patterns and make decisions off of those patterns. Neural networks are one of the more popular forms of machine learning today, despite not being popular in the past. The reason for this is they require heavy computation and long run times which have been enabled by modern computers. In the past the computational resources were not available to make neural networks as effective as they are today. Neural networks are a form of discrete, supervised machine learning. Discrete machine learning attempts to classify input data as one of a discrete number of classes. Supervised machine learning already knows the solution to training data. So in the data that is being used the input value will be the data being classified and the actual value will be the actual answer. Because the output is known, the hypothesis created by the neural network and actual answer can be compared to determine how the neural network needs to be modified. A neural network iteratively computes a set of weights that can be applied to input data to determine its classification. The two techniques used for computing this set of weights are forward and backward propagation. Forward propagation computes the hypothesis on the input data, basically what it expects the input data is classified as. Backward propagation takes this hypothesis and calculates its deviation from the actual classification. Then it adjusts the weights based on this error. The goal is to minimize error and create the best set of weights for classifying new input data. One the Neural network has been trained sufficiently, it can be used to classify input data with unknown output values. Using forward propagation one can compute the hypothesis and assume with some probability that it is correct. The accuracy of a neural network can be computed by taking a set of input data, performing forward propagation, and then taking the ratio of the correct number of classifications versus the total number of classifications.

The actual application of the neural network used in this project is to recognize handwritten digits. This project uses resources from Stanford's machine learning course on *Coursera.com.* While taking this class it was an assignment to implement the neural network in matlab. The training data and the algorithm used in this project are the same training data and algorithm I used in the class. The input data is in the form of 2 csv files. The first csv file , “X.csv”, represents the input images. It is a 5000x400 element matrix, each row (1x400) represents a 20x20 pixel image. The other file “y.csv” represents the images classification. It is a 5000x1 element matrix. Each row (1 element) is a single number (0-9) representing what its respective image (in “X.csv”) actually is. For example row 500 in “X.csv” represents an image of a handdrawn 0, and in “y.csv” row 500 is 0.

As stated before, neural networks require lots of computation. They make heavy use of matrix arithmetic, and use lots of data. There are no dependencies when performing forward and backward propagation on each of these training examples (training example = image), and because of this forward and backward propagation for each training example can be performed in parallel.

Sequential Implementation:

In table 1 (in the appendices) I have included the resulting accuracy and time elapsed for each set of 100 iterations. This run took close to 5000 seconds or about 80 minutes. I currently have the

project set to use initial random matrices I generated a while ago. This means that one should get the same accuracy result for any number of iterations listed below.

This run was done on the discovery cluster. The code was timed using the same method we used in class for the first sequential assignment. At the start of gradient descent I took a start timestamp with:

```
start = clock();
```

And each time I reached a 100 iteration mark I took a stop time stamp and printed the difference:

```
end = clock();
cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
```

The finer grained timing allows graphs to be made for accuracy versus time and so that I can compare runs with lesser iterations to runs with larger iterations and make sure the accuracy and time used is very similar.

2. OpenMP Results:

I chose to use OpenMP for the first parallel implementation of my code. It was clear that a good starting point would be OpenMP or MPI. I chose OpenMP because it uses the shared memory model and will be faster when I am running it on the smaller shared memory nodes. The implementation was very simple and took little over an hour to make. The runtimes and speedups are available in the appendices. At 32 processes a speedup of 12 was achieved, with an efficiency of .38. It is interesting to see that the efficiency of the OpenMP implementation dropped very quickly as more processes were introduced. At 4 processes its efficiency was .86, which means its efficiency dropped 50% between 4 and 32 processes. I did not expect this because each thread is given a significant amount of computation and the time for thread synchronization and reduction should be negligible. One way to get more speedup would be to look into optimizing the reduction and thread synchronization.

I ran the parallel code on the discovery cluster and it was timed using 'omp_get_wtime()'. I simply took an initial time stamp at the start of gradient descent and then subtracted the initial time stamp from timestamps taken every 100 iterations. OpenMP was an effective form of parallelism. Not only did it give a 12 times speedup but it also took only an hour to implement. Part of the reason it was so effective is because the code runs 5000 images through the gradient algorithm and then adds the gradient value to the classifier. The problem is very easy to parallelize because one can just give each thread its own set of images to work on and then at the end perform a reduction to add the gradients of each thread together.

3. Cuda Results:

I chose Cuda as my next form of parallelism because prior to beginning this project I really wanted to try getting awesome speedup on a GPU. Unfortunately the task of converting my sequential code to Cuda proved to be too much work at the time. I had spent roughly twenty hours on the task before realizing that my time was better spent tackling other problems. My issues included transferring non-serialized data structures, dealing with dynamic memory allocations, and figuring out which abstraction layer to apply Cuda programming principles too. One of my structures 'matrix_list_t' contains an array of points to varying sized matrices. While these matrices are serialized their size is not fixed and therefore they cannot be serialized efficiently and prior to being sent they need to be packed and then unpacked upon being received by the GPU. The next problem I ran into was dynamic memory allocation on the GPU, I am not sure whether this problem was actually solved or not. But my attempt at solving it consisted of allocated a huge buffer on the GPU and then casting it to a 'buffer' object that I just incremented each time memory was needed. The last problem I ran into proved to be

the hardest. I was unable to figure out which level I should apply parallelism to. I was certain that if I applied parallelism as low as the matrix arithmetic layer I would see very little speedup, so I wanted to apply it at the same layer I applied it to at OpenMP: the gradient calculation layer. So I wrote the code that each thread got one image and calculated the gradient for that image and then a reduction was performed by block and then by grid. Because there was so much code that needed to be parallelized I ran into multiple issues and it was never completed.

4. Matlab Results

Matlab was a form of parallelism I planned to use because I had originally implemented the code in Matlab. This was very easy to add parallelism to, I simply added a SPMD statement where I had already done so in OpenMP and then divided up the work amongst the processes. I am still working on collecting results on the discovery cluster due to job submission issues. For this reason I have temporary used my Desktop to run the jobs. The results are available in the appendices. A 1.27 speedup was achieved with 4 processes. This efficiency is only .32. I imagine this lack of efficiency at such a low process count is due to my Matlab code not being written well for the SPMD model.

5. MPI Results

MPI was the last form of parallelism I chose and was the one I wanted to do the least. As it turns out it was my favorite and the most interesting. I used the same principle for packing the matrix lists as I had done in Cuda and this solved the memory serialization issue. After this the problem was fairly straightforward all that had to be done was restructure the code a bit to account for the fact that memory did need to be sent to the processes rather than the processes sharing memory. I was able to achieve a 11.55 speedup on 32 processes. These results are available in the appendices. Despite using a distributed memory model and requiring 1000 gathers and 1000 scatters MPI performed 95% as well as OpenMP for 32 threads. This was a huge surprise to be because I was expected the communication overhead to have a much larger impact on the performance of the application. Also the efficiency of MPI scaled much better with the number of processes than OpenMP did. While OpenMP had an efficiency of .86 at 4 processes it dropped all the way to .38 at 32 processes. MPI had an efficiency of .52 at 4 processes and .36 efficiency at 32 processes. For this reason I think it would be very interesting to see how well the MPI implementation scales on multiple nodes. While it is possible to use both MPI and OpenMP together, it would probably not be worth the effort to extract the 5% parallelism when I can continue to just run purely MPI for a larger number of processes.

References and Links:

Unfortunately because one must signup to view the material on Coursera I cannot provide a direct link to the material, but I have found a site that displays in depth notes on the course.
http://www.holehouse.org/mlclass/09_Neural_Networks_Learning.html

The repository for the project is located at:
https://github.com/bcrafton/neuralnetwork
The sequential sub-directory is located at:
https://github.com/bcrafton/NeuralNetwork/tree/master/Sequential
The OpenMP sub-directory is located at:
https://github.com/bcrafton/NeuralNetwork/tree/master/OpenMp
The Cuda sub-directory is located at:
https://github.com/bcrafton/NeuralNetwork/tree/master/Cuda
The Matlab PCT sub-directory is located at:

https://github.com/bcrafton/NeuralNetwork/tree/master/matlab
The MPI sub-directory is located at:
https://github.com/bcrafton/NeuralNetwork/tree/master/MPI

Instructions for running the different implementations are available in the readme files within the respective sub-directory. Bash scripts for running them on the discovery cluster are also located in the respective sub-directory.

Analysis/Review:

After completing the different forms of parallelism it is clear to me that there are some effective and efficient methods for really speeding up a program. Three of the four methods that I tried proved to be effective. Had I been able to build the Cuda code correctly, I am sure that there would have been a solid speedup as well. Implementations had different strengths. Matlab and OpenMP were fairly quick to make, and if the application was going to be run on a shared memory machine then they would be perfect. But for a large distributed system MPI would be great. My MPI implementation had really strong efficiency. It would be very interesting to see how the efficiency would continue to scale if more processes were available.

Two characteristics that I looked at most were speedup and efficiency. Speedup is simply the sequential run time divided by the runtime of the accelerated implementation. Efficiency is the speedup received divided by the amount of resources used (processes). At first I expected MPI to have both lower speedup and efficiency than OpenMP, but have the ability to run on multiple nodes. A shared memory model will always have better speedup and efficiency on the same number of processes. The reason for this is no communication is needed between the threads except for synchronization. Looking at the results (Table 5, Graphs 1 & 2) it is clear that at 32 processes MPI was 95% as strong as OpenMP. This was due to the computation to communication ratio of the application.

There were a couple of things that really surprised me over the course of this project. The first thing was the added difficulty of programming a distributed memory model. Programming with a shared memory model is significantly easier and the development time was much less. Another thing I did not expect was the attainable efficiency of MPI. There are a couple of things I would have done differently. I would have chosen a project that required less code. Having to write a lot of code reduced the amount of time I had to try out optimization on the implementations. Had I actually switched projects I think Cuda would have been doable, but because there was so much code to parallelize it was too hard.

Future Work:

I still need to run my Matlab code on the discovery cluster. For the remaining time I have to work on my project I would like to try making matrix and memory access optimizations in my MPI code and attempting to run MPI on multiple nodes. If I had even more time I would like to actually make the Cuda implementation work and then try out MPI and Cuda together on multiple GPUs because that was my original plan and I believe would yield the largest speedup.

Appendices:

Iteration #	accuracy	Time Used(s)
100	0.761	448.26
200	0.8562	993.88
300	0.8894	1489.12
400	0.9066	1984.85
500	0.9154	2480.73
600	0.9124	2976.03
700	0.925	3473.44
800	0.929	3969.63
900	0.9322	4465.32
1000	0.9358	4961.81

Table 1: Results for Sequential Neural Network

Iteration #	accuracy	time taken (4 processes)	time taken (8 processes)	time taken (16 processes)	time taken (32 processes)
100	0.761	234.14	128.27	79.81	42.13
200	0.8562	476.65	255.31	154.34	84.75
300	0.8894	713.09	386.21	233.86	128.16
400	0.9066	954.71	515.45	310.35	171.33
500	0.9154	1195.04	650.05	389.45	213.5
600	0.9124	1436.54	780.61	468.91	257.25
700	0.925	1675.16	911.22	548.29	300.2
800	0.929	1916.63	1044.25	623.33	343.02
900	0.9322	2158.74	1175.89	704.01	386.7
1000	0.9358	2392.14	1305.94	782.59	429.49

Table 2: Results for Neural Network in OpenMP

Iteration #	accuracy	Time taken (1 Process)	Time used (4 Processes)
100	0.761	47.26	36.9
200	0.8562	93.39	71.77
300	0.8894	138.69	108.44
400	0.9066	184.76	146.06
500	0.9154	232.56	182.33
600	0.9124	278.37	219.26

700	0.925	324.27	254.9
800	0.929	369.47	288.96
900	0.9322	415.97	324.74
1000	0.9358	461.21	362

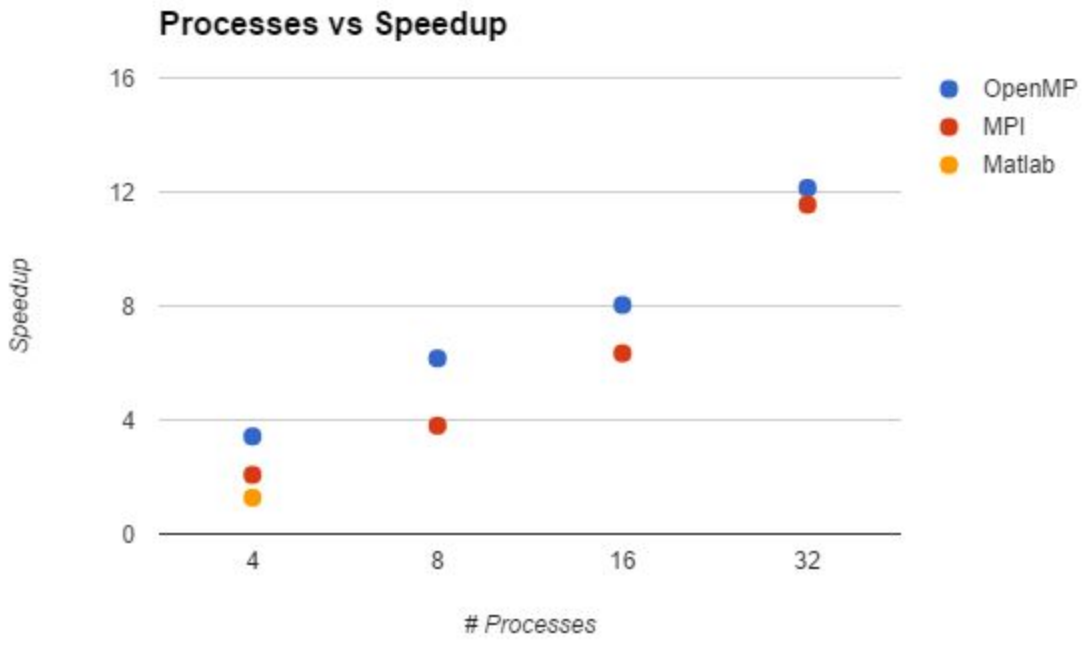
Table 3: Results for Neural Network in Matlab

Iteration #	accuracy	time taken (4 processes)	time taken (8 processes)	time taken (16 processes)	time taken (32 processes)
100	0.761	148.78	90.27	65.96	41.09
200	0.8562	291.61	174.29	125.88	79.25
300	0.8894	437.18	253.66	190.86	122.15
400	0.9066	582.18	329.87	253.79	164.11
500	0.9154	726.76	408.11	317.92	203.15
600	0.9124	870.28	485.73	381.63	246.28
700	0.925	1014.64	562.03	447.92	285.11
800	0.929	1161.18	649.27	507.38	326.91
900	0.9322	1309.61	727.71	561.62	367.93
1000	0.9358	1449.34	805.26	617.43	408.64

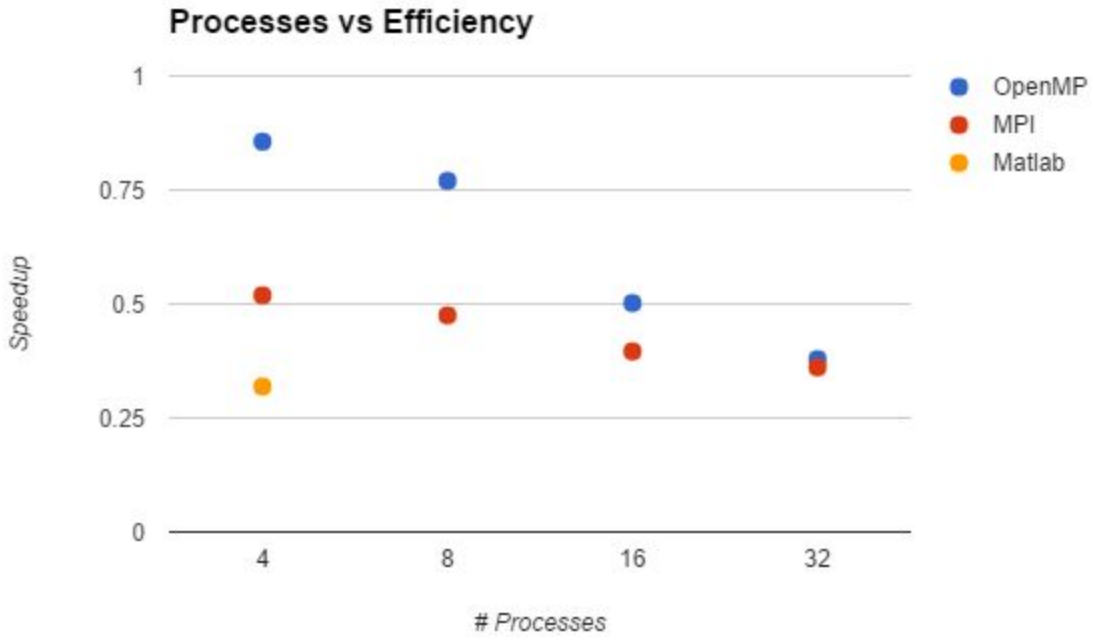
Table 4: Results for Neural Network in MPI

Implementation Type	Processes	Time Taken	Speedup	Efficiency
Sequential	1	4961.81	1	1
OpenMP	4	1449.34	3.424	0.856
OpenMP	8	805.26	6.162	0.77
OpenMP	16	617.43	8.036	0.502
OpenMP	32	408.64	12.142	0.379
MPI	4	2392.14	2.074	0.519
MPI	8	1305.94	3.799	0.475
MPI	16	782.59	6.34	0.396
MPI	32	429.49	11.553	0.361
Matlab	1	461.21	1	1
Matlab	4	362	1.274	0.319

Table 5: Speedup/Efficiency



Graph 1: Speedup Graph



Graph 2: Efficiency Graph