Brian Crafton
High Performance Computing
April 27, 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.

After collecting data for runs of 32 or less processors I tried running 64. The reason for this was seeing how OpenMP performed on more than one node. As expected it performed worse because it relies on a shared memory model and forcing it to split accross nodes is not what it is built for. However it should be noted that the slowdown was not that much. It was roughly 20% slower than the run with 32 nodes.

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. The performance is much worse than expected. The same code that code a speedup on my desktop computer got a slowdown on the discovery cluster. The results are shown in the appendices but it is quite clear that something is wrong with the way work is being allocated or memory bandwidth is being used. It should be noted that the Matlab speedup is shown versus sequential Matlab code. Otherwise the speedup would not be accurate because the Matlab code and sequential C code performed much differently. I believe Matlab may be trying to use a distributed memory model under the hood despite only a shared memory model being needed. If this were the case and it was recopying data that does not need to be copied each iteration, then it is likely that the code would get slower as the number of processes increased.

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 and a 26.22 speedup on 144. 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.

I gave MPI special attention with running on multiple nodes and large process numbers because it can scale well. After trying the MPI implementation with large numbers of processes (128, 256, 512) I was able to achieve over a 25 time speedup. The efficiency did go down quite a bit the there is a clear reason for this. The data set includes only 5000 images and when the data set is sent to

large numbers of processors the communication time becomes larger than the actual computation time. Had the data set been larger, then larger numbers of processes (256, 512) would certainly do better than smaller numbers of processes. Every time the gradient is calculated the master processor has to send data to each slave processor, let it do the computation, and then collect the results. When the number of images sent becomes smaller the computation time scales worse than the communication time. For this reason when the number of processes increase past a certain point (144 in this case) the communication time has become more expensive then the computation time.

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 maintained a 20% efficiency with 144 processes and would definitely have higher efficiency if there were more data.

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 6, 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.

It is inspiring to see how well HPC can perform on these types of problems. With the speedups I was able to achieve I was able to run my code for far more iterations in less time. The neural network maxed out at 99.7% accuracy, but this was only achievable with a parallelized implementation. It took the sequential code 14 hours to achieve a 98.9% accuracy. This is 39 more correctly identified images. This could apply to other real world applications where machine learning is being used where high accuracies are required. It would be extremely frustrating if the software one was using to identify handwritten digits was to incorrectly identify so many images!

Future Work:

Future work on this project would include optimizing matrix arithmetic for memory accesses, and trying to implement Cuda. This problem would work very well on a GPU but unfortunately it is very difficult to program on one.

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: Sequential Implementation

Processes	Time Taken	Speedup	Efficiency
4	1449.34	3.42	0.86
8	805.26	6.16	0.77
16	617.43	8.04	0.5
32	408.64	12.14	0.38
64	529.8	9.37	0.15

Table 3: MPI Results

Processes	Time Taken	Speedup	Efficiency
1	1,616.48	1	1
4	1,116.10	1.45	0.36
8	1035.1	1.56	0.2
16	1095.9	1.48	0.09
30	1695.4	0.95	0.03

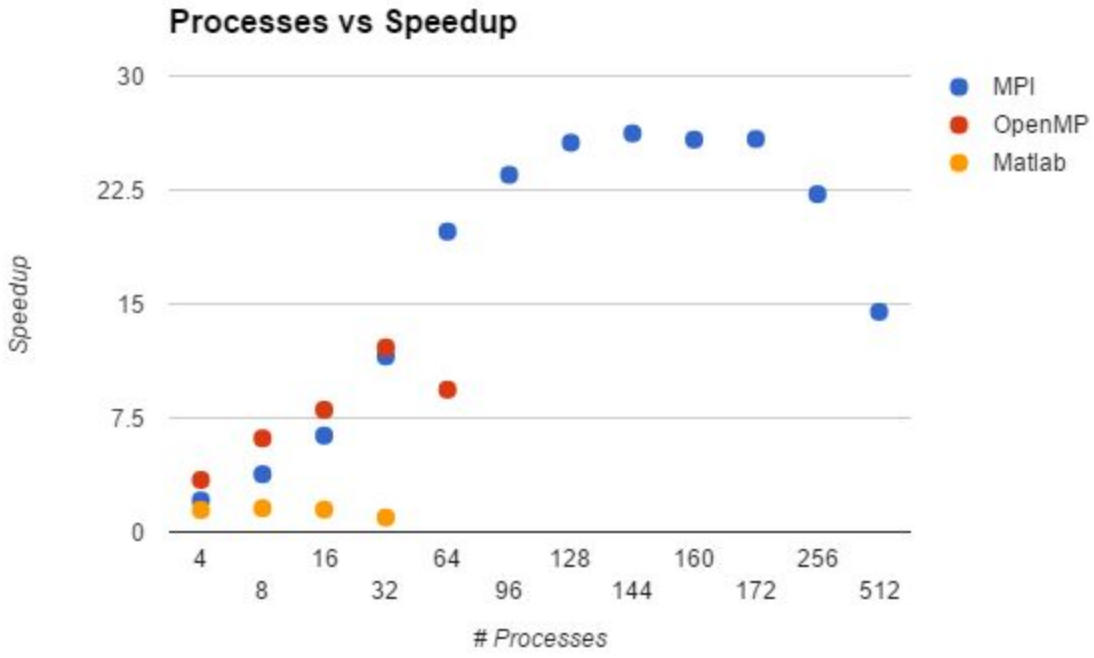
Table 4: Matlab Results

Processes	Time Taken (1000 iterations)	Speedup	Efficiency
4	2392.14	2.07	0.52
8	1305.94	3.8	0.47
16	782.59	6.34	0.4
32	429.49	11.55	0.36
64	251.03	19.77	0.31
96	211.18	23.5	0.24
128	193.75	25.61	0.2
144	189.22	26.22	0.18
160	192.19	25.82	0.16
172	191.88	25.86	0.15
256	223.26	22.22	0.09
512	342.43	14.49	0.03

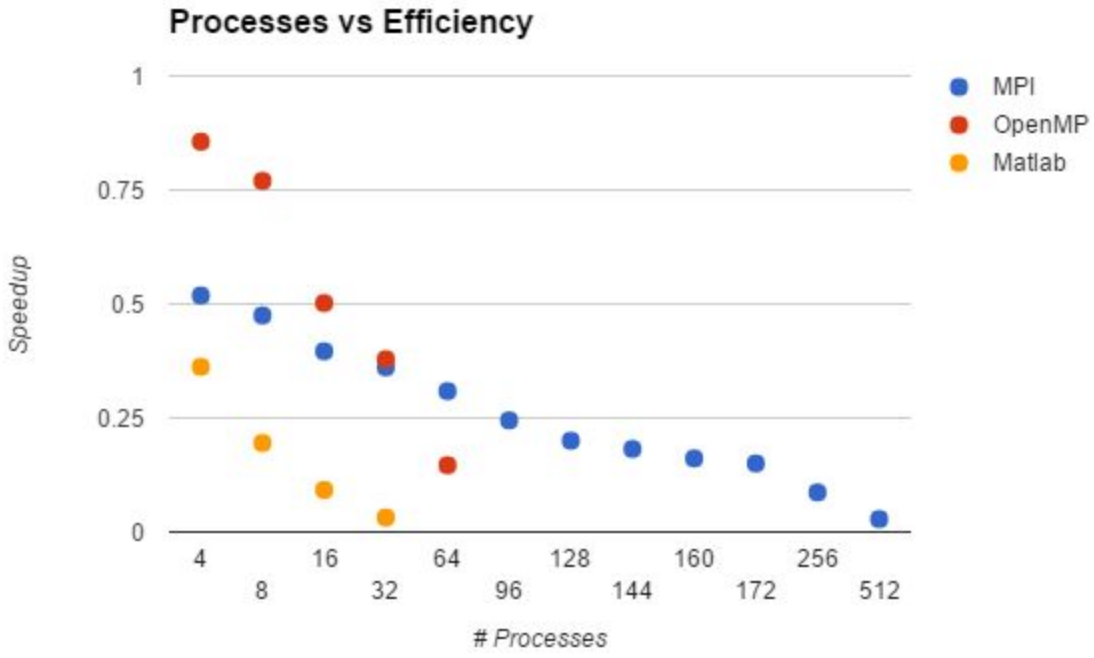
Table 5: MPI Results

Implementation Type	Processes	Time Taken	Speedup	Efficiency
Sequential	1	4961.81	1	1
OpenMP	4	1449.34	3.42	0.86
OpenMP	8	805.26	6.16	0.77
OpenMP	16	617.43	8.04	0.5
OpenMP	32	408.64	12.14	0.38
OpenMP	64	529.798389	9.37	0.15
MPI	4	2392.14	2.07	0.52
MPI	8	1305.94	3.8	0.48
MPI	16	782.59	6.34	0.4
MPI	32	429.49	11.55	0.36
MPI	64	251.029151	19.77	0.31
MPI	96	211.183593	23.5	0.24
MPI	128	193.750761	25.61	0.2
MPI	144	189.219384	26.22	0.18
MPI	160	192.189011	25.82	0.16
MPI	172	191.883948	25.86	0.15
MPI	256	223.255304	22.22	0.09
MPI	512	342.427069	14.49	0.03
Implementation Type	Processes	Time Taken	Speedup	Efficiency
Matlab	1	1,616.48	1	1
Matlab	4	1,116.10	1.45	0.36
Matlab	8	1035.1	1.56	0.2
Matlab	16	1095.9	1.48	0.09
Matlab	30	1695.4	0.95	0.03

Table 6: All results
Note: Matlab speedup and efficiency is based on sequential matlab code



Graph 1: Speedup Graph



Graph 2: Efficiency Graph