Brian Crafton
High Performance Computing
March 18, 2016

Parallelizing A Neural Network

Introduction:

Neural networks are a popular form on machine learning used for classifying data. They use a forward and back propagation technique to compute a gradient and a cost. The gradient is a computed value of the direction of the solution and the cost is the scaled "failure" of the current solution. Using these two values computed using forward and back propagation an optimization method can find a local minimum solution. By fine tuning the value for the learning parameter the programmer can hit a good local minimum rather than one that is not a very good solution. Machine learning uses statistics to give accurate predictions based on training data. A neural network is a supervised, discrete implementation of machine learning. Supervised machine learning uses training data that has already been classified. Discrete machine learning classifies the input data as a class or category as opposed to giving a continuous prediction. In machine learning it is important to be able to gauge the accuracy of the classifier. Fortunately is is very simple to measure the accuracy of a neural network as one can simply take the largest probability from their classifier and check if it maps to the correct value.

Neural networks require lots of matrix arithmetic and lots of repetitive steps. Matrix arithmetic requires lots of independent instructions, making it ideal for parallel computing. Operations like matrix addition, matrix transpose, matrix multiplication, matrix splitting, ect are all computationally intensive operations that have lots of (embarrassingly) parallel operations. This type of parallelism is data parallelism because the same computation is being done just on different data. To exploit data parallelism I plan to use SIMD/SIMT and multi-threading.

This project is based on the neural network section (weeks 4 and 5) in Stanford's machine learning course on *Coursera.com*. The application for this project is classifying hand written digits, so taking 20x20 pixel (greyscale, uncompressed) images and classifying each one as a digit (0-9). The data I am using is a 5000x400 matrix which I have taken from Stanford's Machine learning course on *Coursera.com*. Each row (400 double floating point values) represents a 20x20 greyscale image that is a handwritten digit. I completed the class assignment in Matlab, but the implementation for this project is written in C, specifically written for the gnu gcc c compiler. There are two intentional design flaws from the start.

1. I have written basic implementations for the matrix arithmetic from scratch
2. I am using gradient descent rather than using a popular optimization function

The reason for these design choices is because the assignment is about learning rather than building an effective neural network. Success will be determined by the parallel speed up achieved rather than the actual accuracy of the neural network. There are two high level techniques that will be investigated.

1. Paralleling the matrix arithmetic
2. Giving each thread its own neural network iteration in the neural network and have each thread perform sequentially and come back together to compute the new classifier.

There will be tradeoffs between both, but after doing some trial and error I am hoping that the better technique will be clear.

References and Links:

Unfortunately because one must sign up 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

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

Results:

1. Sequential Results:

In table 1 (in the appendices) I have included the resulting accuracy and time elapsed for each set of 100 iterations. This run took roughly an hour and used close to 5000 seconds or about 80 minutes. I currently have the project set to use initial random matrices I generated awhile 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 time stamp 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. At first I tried 16 threads and then switched to 32 threads. For 16 threads the speedup I received was roughly 8. When I ran 32 threads the speedup when up to 12. The speedup values and the run times for the sequential and OpenMP runs are in the appendices section.

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 time stamps 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.

Next Step:

The next form of parallelism I plan to implement is Cuda.

Appendices:

Iteration #	accuracy	Time used (s)
100	.7610	448.26
200	.8562	993.88
300	.8894	1489.12
400	.9066	1984.85
500	.9154	2480.73
600	.9124	2976.03
700	.9250	3473.44
800	.9290	3969.63
900	.9322	4465.32
1000	.9358	4961.81

Table 1: Results for Sequential Neural Network

Iteration #	accuracy	Time used (s)
100	.7610	58.927110
200	.8562	116.912429
300	.8894	179.397450
400	.9066	241.980347
500	.9154	297.289525
600	.9124	357.543150
700	.9250	421.107535
800	.9290	483.281073
900	.9322	543.077274
1000	.9358	606.274746

Table 2: Results for Neural Network in Open MP (16 Threads)

Iteration #	accuracy	Time used (s)
100	.7610	40.012822
200	.8562	83.094407
300	.8894	123.597567
400	.9066	164.134006
500	.9154	206.178307
600	.9124	247.474300
700	.9250	287.893888
800	.9290	331.332767
900	.9322	372.466834
1000	.9358	413.329154

Table 3: Results for Neural Network in Open MP (32 Threads)

Implementation	# threads/processes	Time for 1000 iterations	Speedup vs Sequential
Sequential	1	4961.81	1
OpenMP	16	606.274746	8.18
OpenMP	32	413.329154	12.00

Table 4: Speedup vs Sequential Code