Kobe Davis

Prof. Doliotis

CS 445

13 March 2020

Comparison of Parallel & Sequential Multi-Layer Neural Network Performance

Introduction

The goal of this project is to conduct a performance/scaling analysis and comparison between three different implementations of the Multi-Layer Neural Network (MLNN) algorithm discussed in class. The implementations to be compared are in:

1. Python using Numpy
2. C++ using OpenMP
3. C++ using Eigen

The MLNN algorithm for these implementations is the same algorithm covered in class and used in assignment 2. The Python implementation is simply the code for assignment 2 (minus extra formatting and confusion matrix sections). The OpenMP implementation focuses on parallelizing the feedforward, error, and backpropagation stages of the program. Matrix multiplication as well as all other operations involved in these stages are rewritten manually as loops so as to take advantage of OpenMP parallelism (as opposed to relying on Eigen's inherent use of OpenMP, to that degree OpenMP is disabled for Eigen explicitly). Lastly, the Eigen implementation relies fully on the Eigen API for performance benefits and parallelism. Due to the fact that Eigen utilizes OpenMP for its own algorithms, OpenMP must be included in the compilation of this implementation. Though no OpenMP compiler directives are used anywhere within the Eigen implementation.

I fully expect both the OpenMP and Eigen implementations to deliver substantially better results than the Python implementation. Despite the added performance boost that Numpy contributes to the Python MLNN implementation, the program is still sequential and Python remains an accessible yet sluggish language. Rewriting the program in C++ should cut down on execution time well enough, but then there are a plethora of matrix operations just waiting to be parallelized. Additionally, I would speculate that the Eigen implementation will outperform the OpenMP implementation for the sole reason that the Eigen library is heavily optimized and has been through many iterations, whereas the OpenMP implementation will prioritize correctness; as I will be rewriting many matrix and vector operations from scratch, and that introduces the potential for error.

Materials and Methods

All measurements were performed on the Particle Lab systems at Portland State University. Particle Lab systems contain identical hardware, hence the following specs represent all systems involved in performance measurements: Architecture: x86_64, CPU Model: Intel Xeon E3-1241v3 3.5GHz, Cores: 4, Threads/Core: 2.

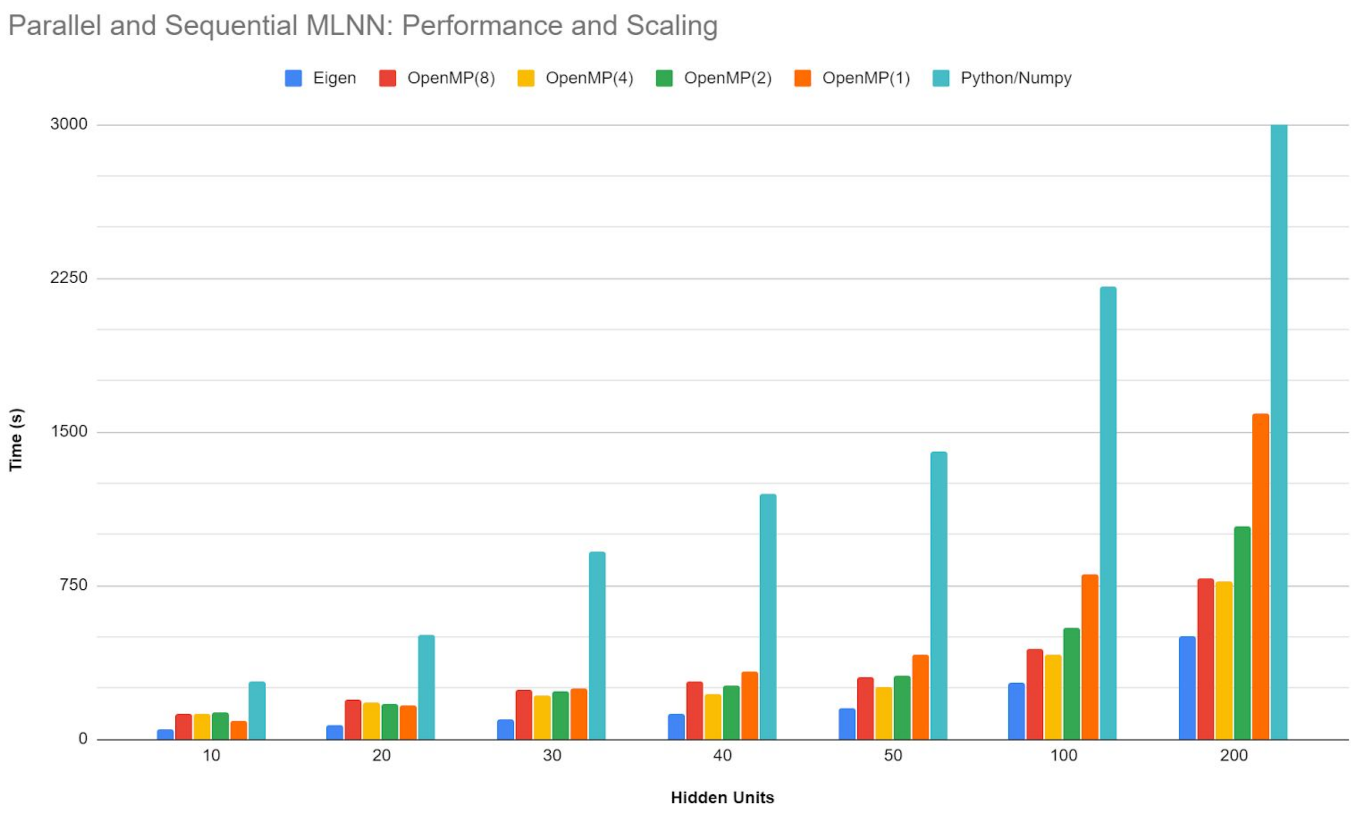
Eight threads is the maximum number of threads available to the Particle Lab systems, and while Babbage has significantly more cores and threads the single core performance was bad enough that the Particle Lab systems outperformed it by a large margin. This made the Particle Lab systems a good choice for this project.

The primary metric for performance used in this paper is execution time. Number of hidden units was used as the independent variable, and covered the following values during measurements: 10, 20, 30, 40, 50, 100, and 200 hidden units. For each number of hidden units, the mean of five runs for each implementation was taken as the execution time. This is straightforward for both the Python/Numpy implementation and the Eigen implementation, though for the OpenMP implementation I attributed execution times to the number of threads used. This decision resulted in four categories for execution times under the OpenMP implementation: 1 thread, 2 threads, 4 threads, and 8 threads. As for any parallel program, the number of threads should heavily affect the execution times, so this methodology presents a sliding scale from sequential performance (1 thread version) to parallel performance (8 threads).

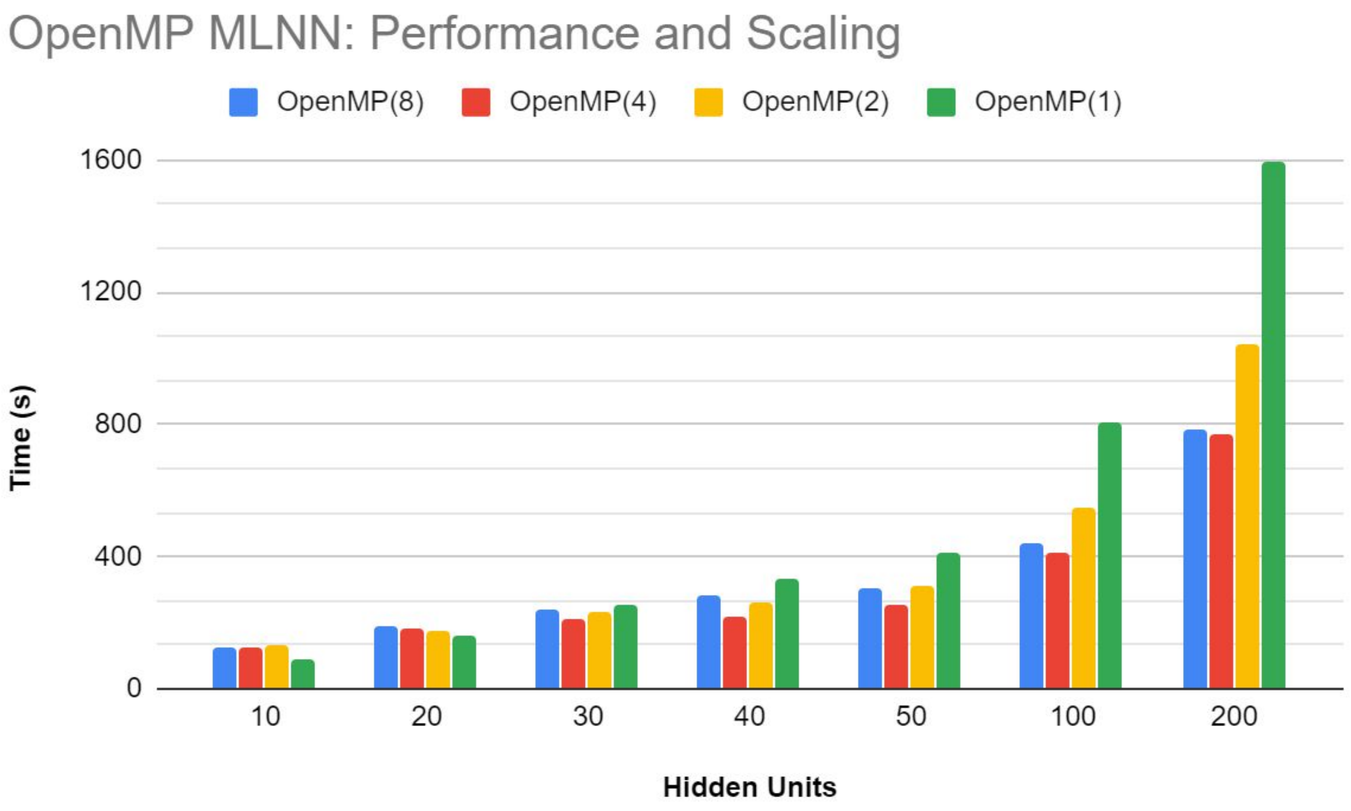
All together, six sets of measurements were taken and each set contained seven measurements (six categories, seven different numbers of hidden units). Considering the five runs performed to obtain an average for execution time, 210 measurements were performed in total. Neural nets are not the fastest running programs so this definitely took some time.

Results

In reference to the first image below: of the three implementations, Eigen did exceptionally well. The OpenMP implementations did as expected, better than Python but not quite as good as pure Eigen. And at 200 hidden units the Python/Numpy execution actually had to be cut off in the following image to preserve the visibility of the chart; the execution time reached roughly an hour (3,994 seconds). With that said, the Python/Numpy implementation keeps pace to a certain extent during the lower hidden unit counts. The superior performance of the C++ implementations becomes apparent starting at 30 hidden units. The superior scaling of the parallel implementations becomes apparent at 100 hidden units. Thereafter both the sequential C++ (OpenMP (1)) and Python/Numpy execution times diverge rapidly.



In reference to the second image: while the sequential and 2-thread versions appear to improve by increasing the number of threads, the 4 and 8-thread versions do not change much from one to the other. All parallel versions of the OpenMP implementation appear to remain bunched together despite the growing number of hidden units. Interestingly, the overhead from parallelizing the code is actually visible at 10 and 20 hidden units. And since the plotted execution times are averages, it likely means that the sequential C++ version consistently outperformed its threaded OpenMP counterparts at lower hidden unit counts.



An expected outcome is seen in the improved scaling of all parallel implementations. While all implementations reach similar execution times at lower hidden unit counts, we can see the growth of the sequential execution times slowly diverge from their parallel counterparts. Even in the well performing sequential C++ version (OpenMP (1)), it is still apparent that its execution time begins to pull away from the parallel versions, alongside the Python/Numpy execution time. Given more time, larger hidden unit counts could be plotted and the scaling strength of the parallel implementations would likely become more apparent.

Performance measurements for these implementations are included in the appendix at the end.

Discussion

This outcome is in line with my initial expectations, though there are a few surprises. The first being that simply rewriting the MLNN in C++ caused such a large gap between Python/Numpy execution times and the newer implementations' execution times. Additionally, the surprisingly good scaling of the sequential C++ version (OpenMP (1)) was somewhat unexpected. While it is clearly lagging behind the others as the hidden unit count grows, it still maintains a comfortable distance between itself and the Python/Numpy execution time. In short, better outright performance was expected of the sequential version but not necessarily better scaling, hence the surprise.

The outcome I found most surprising was the relative performance between the varied threads in the OpenMP implementation. At every measurement besides the first, the 4-thread version performs *better* than the 8-thread version. This was interesting because while it is generally true that increasing the number of threads will result in diminishing returns at a particular point, its unusual that a small increase in threads resulted in consistently worse performance where it did not previously in the jump from 2 to 4 threads.

For future work it would be interesting to divide the MLNN code into sections that would be measured separately from each other. This work could validate answers to questions such as: What percentage of the MLNN code is parallelizable? What is the theoretical maximum speedup achievable by parallelizing the MLNN code? Are the parallelizable sections of the MLNN code also necessarily bottlenecks, or are certain inherently sequential sections the primary factor in the MLNN execution time?

I believe these questions are important because knowing where to target an algorithm is the most important part in making an algorithm parallel. This idea goes hand-in-hand with avoiding premature optimization. With that said, there are plenty of matrix and vector operations in the MLNN algorithm, which are always easy targets for parallelization. After completing this project, I'm curious to know more about how modern matrix multiplication optimizations are combined with parallelization techniques to obtain even better performance. I imagine this is something the Eigen library has implemented from its outstanding performance presented in this paper.

Acknowledgements

The C++ linear algebra library, Eigen, was used to help implement the MLNN algorithm. All other code is from the C++ standard library and was written by me.

Appendix

Eigen	
# of Hidden Units	**Execution Time (s)**
10	46.94
20	65.99
30	96.33
40	123.71
50	148.40
100	274.72
200	502.34
250	618.45
300	760.28
350	862.68
400	989.97

Python/Numpy	
# of Hidden Units	**Execution Time (s)**
10	283.97
20	511.31
30	916.37
40	1196.07
50	1407.41
100	2207.97
200	3994.83

OpenMP (1)	
# of Hidden Units	**Execution Time (s)**
10	86.98
20	164.44
30	251.18
40	330.05
50	412.09
100	804.32
200	1591.32

OpenMP (2)	
# of Hidden Units	**Execution Time (s)**
10	131.71
20	173.8
30	232.96
40	262.46
50	309.23
100	546.14
200	1040.04

OpenMP (4)	
# of Hidden Units	**Execution Time (s)**
10	122.88
20	180.26
30	213.41
40	218.44
50	254.95
100	414.97
200	767.55

OpenMP (8)	
# of Hidden Units	**Execution Time (s)**
10	122.93
20	189.41
30	238.35
40	279.98
50	318.37
100	437.22
200	783.27