Parallelization of a Neural Network for Handwritten Digit Recognition Using OpenMP

1. Introduction

This project focuses on optimizing the execution time of a neural network that classifies handwritten digits. The neural network has already been provided to us by the professors, written in C. Our one and only goal is to apply parallelization techniques, using OpenMP, to not only make the code faster, but also maintain the accuracy of the original sequential version.

To do so we have been given access to 2 different clusters, Aolin (faster) and Wilma (quicker), I however, decided to develop the whole project using this last one (more concretely the nodo.q partition), since I run into several issues using the first one. However, I believe that results will be equally good when using all other clusters or partitions.

2. Methodology

To begin this project, I started off by saving both the original training.c file ("training_og.c") and the initial executions, both with the -fopenmp instruction and without it (purely sequential). This was mandatory since a reference point was needed, as well as keeping a virgin version of the precise file the parallelization had to take place. I could then have a couple times to compare my results to (besides the ones proposed by the professor), and a version I could go back to.

Also, I decided to work from the initial default configurations (10 epochs) and once I made advancements I would test them with also 1 and 100 epochs. I also decided to work function by function since that was what I believed to be optimal.

feed_input() - The feed_input() function, which simply copies an input image into the input layer of the neural network, wasn't parallelized because it's extremely fast and doesn't have any noticeable impact on overall performance, since running it sequentially takes almost no time.

Trying to parallelize it would actually add unnecessary overhead from OpenMP thread setup, likely making it slower rather than faster. Because of this, and since the function isn't a bottleneck, we chose to keep it simple and leave it sequential. However, I tried parallelizing it and my initial hypothesis was confirmed so I moved onto the next function.

forward_prop() - The forward_prop() function is the part of the neural network where input values are propagated through each layer to compute activations. Since each neuron's activation in a given layer is calculated independently of the others, this was a perfect spot for parallelization. We added an OpenMP **"#pragma omp parallel for" (line 51)** directive before the inner loop over neurons to distribute the work across multiple threads. This allowed each thread to handle a subset of neurons, speeding up the forward pass.

This change worked really well because there were no dependencies between the neurons during this step, so the parallelism was clean and didn't introduce race conditions. It also gave us a noticeable speedup without affecting the correctness of the outputs, since the activation calculations remained exactly the same as in the sequential version.

back_prop() - The back_prop() function is where the neural network calculates the gradients needed to adjust the weights and biases after each training example. It's more complex than the forward pass because some of the values depend on others, which makes parallelizing it a bit more delicate.

We started by parallelizing the loop that processes the output layer using **"#pragma omp parallel for" (line 90).** Each output neuron computes its own gradient independently, so this was safe and gave us a bit of a boost.

The next part — updating the gradients for the hidden layers — required more care. We parallelized the main loop over neurons using another **"#pragma omp parallel for" (line 111),** but inside that, there's a shared array (dactv) that multiple threads might try to update at the same time. To avoid race conditions, we protected that specific update with a **"#pragma omp critical" (line 120),** ensuring only one thread at a time modifies that value.

I also tried to parallelize the middle loop, however, no matter what I did I couldn't manage to parallelize it properly due to the race condition, I tried "critical" amongst other directives and nothing seemed to work.

Even though this parallelization was barely noticeable, I kept it because, not only wasn't altering the accuracy, but it was slightly quicker than execution where we added the "-fopenmp" flag, which meant that it was actually working and it proved to be the right decision later.

update_weights() - The update_weights() function was one of the simplest parts to parallelize. It updates the weights and biases of the network using the gradients calculated during backpropagation. Since each weight and bias is updated independently, we were able to safely parallelize both loops using **"#pragma omp parallel for" (lines 141 and 151)**. There were no race conditions or shared data conflicts, making this a very clean and efficient optimization.

This change, once again, wasn't better than the original sequential execution, but similar to the last one it was better than the sequential execution that used "-fopenmp" which also meant that parallelization was working.

Combining functions

If my hypothesis was right, and all previous functions really lead to parallelization, once I combined them I would get better results, since combining multiple parallelized functions should lead to significantly better performance because it reduces the overall amount of sequential work in the program. When only one function is parallelized, the others still run sequentially and act as bottlenecks, limiting the potential speedup according to Amdahl's Law.

However, when forward_prop(), back_prop(), and update_weights() are all parallelized, the program is able to distribute most of the computational workload across multiple threads. This keeps all available CPU cores busy for longer, reduces idle time, and maximizes the efficiency of parallel execution. As seen in the results down below.

This table is extracted from the "results" folder in the github repository, more precisely inside of "**results/wilma_nodoq/testing**".

Wilma (nodo.q)			
All with 8 threads	**TESTING TIME**		
Functions Parallelized	1 Epoch	10 Epochs	100 Epochs
Sequential	1.258692 sec	10.510997 sec	112.238939 sec
OpenMP directive, no parallelization	1.540723 sec	12.539685 sec	129.287405 sec
forward_prop()	1.068450 sec	10.423450 sec	100.056324 sec
back_prop()	1.319639 sec	11.037407 sec	107.959326 sec
update_weights()	1.335702 sec	10.981486 sec	109.749137 sec
forward_prop() and back_prop()	1.093288 sec	10.456663 sec	103.198880 sec
forward_prop()and update_weights()	0.812514 sec	8.058950 sec	70.181922 sec
back_prop() and update_weights()	0.972437 sec	7.501003 sec	72.528562 sec
forward_prop(), back_prop() and update_weights()	0.290060 sec	2.286927 sec	22.273747 sec

3. Final results and conclusions

This tables down below prove the effectiveness of my parallelization. I used both "Wilma's nodo.q" and "Aolin's test.q" partitions for this final results and created this tables for easier interpretation. I ended up using only 4 and 8 threads both guided by the professor's "Reference Times" as well as for sake of simplicity. (I also tested more threads but it proved to worsen performance so I decided to not keep them). Also I didn't include any of the results since they were the same as in sequential (1 epoch: 787, 10 epochs: 885 and 100 epochs: 903)

Wilma (nodo.q)			
	FINAL TIME		
Type	1 Epoch	10 Epochs	100 Epochs
Sequential	1.258692 sec	10.510997 sec	112.238939 sec
OpenMP directive, no parallelization	1.540723 sec	12.539685 sec	129.287405 sec
4 Threads	0.422224 sec	3.411978 sec	33.223662 sec
8 Threads	0.290060 sec	2.286927 sec	22.273747 sec

Speedup			
Nº of threads	**1 Epoch**	**10 Epochs**	**100 Epochs**
4 Threads	2.981100	3.080617	3.378283
8 Threads	4.339419	4.596123	5.039068

Efficiency			
Nº of threads	**1 Epoch**	**10 Epochs**	**100 Epochs**
4 Threads	0.745275	0.770154	0.844571
8 Threads	0.542427	0.574515	0.629884

Aolin (test.q)			
	FINAL TIME		
Type	1 Epoch	10 Epochs	100 Epochs
Sequential	0.284192 sec	2.232792 sec	21.909734 sec
OpenMP directive, no parallelization	0.286619 sec	2.242090 sec	21.818904 sec
4 Threads	0.098653 sec	0.714963 sec	6.908835 sec
8 Threads	0.085262 sec	0.612918 sec	5.883015 sec

Speedup			
Nº of threads	**1 Epoch**	**10 Epochs**	**100 Epochs**
4 Threads	2.880723	3.122947	3.171263
8 Threads	3.333161	3.642889	3.724234

Efficiency			
Nº of threads	**1 Epoch**	**10 Epochs**	**100 Epochs**
4 Threads	0.720181	0.780727	0.792816
8 Threads	0.416645	0.455361	0.465529

As we can clearly see the biggest speedup was achieved when using a bigger number of epochs as well as 8 threads, and using less threads obviously lead to a higher efficiency. I would have liked to try it out with more epochs and threads but for the sake of simplicity, and since you already provided some reference times, I believe I have done a good job.

4. Appendix

Since this project has been carried out after the partial exam, we haven't had the opportunity to solve the issues regarding the GitHub space, which I decided to leave to my ex-groupmate since he had already worked in there. As an alternative I created the following GitHub Repository to work on my own implementation: "https://github.com/PolPuigPuy/OpenMP_Project_PolPuig". I hope we can fix that issue soon.

Also, I would like to point out that I've mostly done this project on the last few days (you can check the modification dates in the GitHub), even though I'm not proud of it, it was to prove a point. It's true that I barely worked when working with Davide on this project, since I had to work on other subjects with earlier deliveries, and was planning on doing it in the last moment.

I also understand that it might not be pleasant for other people my way of working, however, I hadn't given him any reasons to believe I wasn't going to do anything. All the previous deliveries had been handed in by me, deliveries in which I had let him choose what exercises he wanted to do and I'd simply do the rest and deliver it. Most times leaving me with the harder exercises or even the whole task for his personal reasons I can understand.

With this I don't want to blame him, I want to defend myself from the accusations that I don't work, since they're not true, and I would have gladly appreciated him not going behind my back and complaining to me face to face.

I understand you might not like my forms but I'd rather you judge me for the results, which I believed to be good.

Lastly, since I couldn't do anything in the verification part of the exam, since the email made me not work in the project and focus more into studying, I would be grateful if there was any way for this project to count for the final mark, and have a chance to pass the subject. Thank you in advanced.