CS 581: High Performance Computing
Emory University
Spring 2025
Prof. Tianshi Xu

Homework # 1

Mitchell Scott
(mtscot4)

1 Environment

1. **List the environment you used to run the code (Operating System, CPU, RAM, etc.)** .

   Solution. The local environment where I was running the code was on a 2022 MacBook Pro with Apple M2 chip, 8 GB of RAM running macOS Ventura 13.3.

2 Sequential Code

1. The performance accuracy for all code variants was **98.14%**.
2. The results are seen in Tab. 1.

Trial	Time (ms)
1	3247
2	3268
3	3288 (**high**)
4	3238 (**low**)
5	3273
Avg	3262.7

Table 1: Sequential Code for model.

3 OpenMP

1. The performance accuracy for all code variants was **98.14%**.
2. The results are seen in Tab. 2. In terms of the speedup, first we need to recall that

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{1}$$

   So simply by dividing our times from the result in section 1, the speedup is seen in Tab. 3.

3. The speedups for sequential (baseline), OpenMP, and PThreads are plotted in Fig. 1.

Trial	1 thread (ms)	2 thread (ms)	4 thread (ms)	8 thread (ms)
1	3399	1768	1015	710
2	3555 (**high**)	1837 (**high**)	924	708
3	3397	1770	917 (**low**)	708
4	3407	1765 (**low**)	1035 (**high**)	702 (**low**)
5	3391 (**low**)	1797	918	745 (**high**)
Avg	3401	1778.3	952.3	708.7

Table 2: OpenMP Code for model over changing thread numbers.

# Thread	Speedup
1	0.95934
2	1.83473
4	3.42613
8	4.60378

Table 3: Speedup of OpenMP for varying number of threads.

4 Pthreads

1. The performance accuracy for all code variants was **98.15%**.

Trial	1 thread (ms)	2 thread (ms)	4 thread (ms)	8 thread (ms)
1	3362 (**high**)	1694 (**high**)	866(**low**)	640 (**high**)
2	3179	1680	1074 (**high**)	664
3	3128 (**low**)	1653 (**low**)	939	646
4	3175	1685	880	665
5	3138	1664	870	683 (**high**)
Avg	3163	1676.3	896.3	658.3

Table 4: PThread Code for model over changing thread numbers.

2. Recalling Eq. 1, simply by dividing our times seen in Tab. 4 by the result in section 1, the speedup is seen in Tab. 5.

3. This has already been done in Fig. 1.

5 Implementation

1. How do you design your sequential implementation to be cache efficient?

   Solution. The forward pass of the algorithm is essentially solving

$$\mathbf{x}_{\text{output}} = \sigma\left(\mathbf{W}\mathbf{x}_{\text{input}} + \mathbf{b}\right), \quad \text{where} \tag{2}$$

$$\sigma(x) = \begin{cases} x, & x \geqslant 0 \\ 0, & x < 0 \end{cases} \tag{3}$$

   is the "rectified linear unit" (ReLU). The $\mathbf{x}_{\text{output}}$ is stored as a long vector of length "output_dim $\times$ batch_size", but can be thought of as a matrix stored F (as in

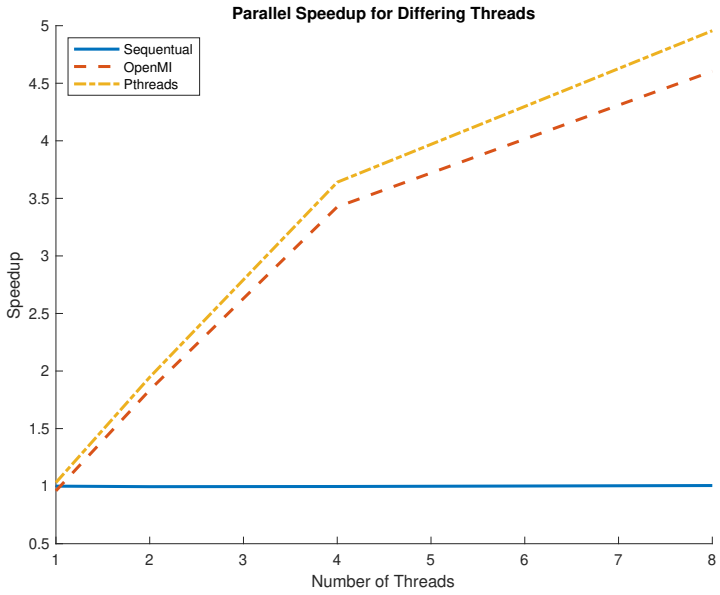

Figure 1: The speedup compared to the sequential code for OpenMP and PThreads for varying threads.

# Thread	Speedup
1	1.03152
2	1.94637
4	3.64019
8	4.95625

Table 5: Speedup of PThreads for varying number of threads.

Fortran) style. Similarly $\mathbf{x}_{\text{input}}$ has the same vector to matrix mapping but with vector of length "input_dim × batch_size". Conversely, $\mathbf{W}$ is a matrix stored as a vector of length "input_dim × output_dim", but it is stored C style, which means that performing the repeated matvec of $\mathbf{W}\mathbf{x}_{\text{input}}$ is essentially an inner product.

First we initialize $\mathbf{x}_{\text{output}}$ as a vector of floats. Then we move $\mathbf{b}$ into the cache to make repeated copies of it into the newly initialized (to zero) $\mathbf{x}_{\text{output}}$. Now that we have accounted for the bias term, we need to increment the matrix-matrix multiplication. We do this by first considering the batches, then the output, then the input. We utilize the fact that $\mathbf{W}$ is C style and $\mathbf{x}_{\text{input}}$ is F style to increment the counters to be cache efficient.

2. How do you parallelize the linear layer and the ReLU activation function?

   Solution. For the OpenMP code, it was very straightforward to turn my model code into parallel code for the linear layer and ReLU function. I simply put the #pragma omp for outside of the largest for loop. Since we are trying to parallelize nested for loops, OpenMP cleverly divies the work up behind the scenes. Since I didn't put this parallel code in the inner more for loop, we didn't have to worry about mutexes or critical regions.

   For the PThreads code, I once again divided up the outer-most for loop to make sure that there was no need for a mutex. Since the outer for loop was over the

batches, I essentially divided the total number of batches by the number of threads to determine the start and end batches that each thread would do. On the last thread, I changed the end batches to the last batch to ensure that no batch was left undone. Since the batches are independent, the concurrent reads and writes were essentially independent processes.

3. What is the performance bottleneck of your parallel implementation?

   Solution. In an ideal world, by doubling the amount of threads that we have, the speedup would be twice as fast. However, this is not the case. This is a demonstration of the communication bottleneck. By having more threads, we still need scatter the data from the cache to new places, and while the heavy computation can be done in parallel, the data still needs to be gathered. This communication is not for free, and is why the speedup will eventually level off.

   Another performance bottleneck might be the load imbalance. Assume we have 15 tasks to perform on 4 processors. Each task would need to perform 3.75 tasks, but in integer division, we would have the tasks performed per processor be 3,3,3,6. This means that the last processor has to do twice the amount of work compared to the other processors. Since the entire process needs to have all processors complete, this would obviously cause a performance bottleneck.

Acknowledgements

I would like to acknowledge that I worked with fellow CS 581 students Emma Hart in OH on 2-4-25. Additionally I did use the Github Copilot to expedite my code (of course I checked the work it produced before I ran it). Lastly, I use resources from the University of Michigan EECS department to figure out how to structure my PThread code.