Assignment #1, MACS 30123

Professor Jon Clindaniel

Submitted by Junho Choi

The "Coding Appendix" to this file is `HW1_junhoc_coding.ipynb`.

Before trying to answer the below questions, I will briefly explain my logic for the matrix (or vector) operation used in the code for this assignment. From the initialization, we know that $z_0 = \mu$. It follows that

$$z_1 = \rho \underbrace{\mu}_{=z_0} + (1-\rho)\mu + \varepsilon_1 = \mu + \varepsilon_1$$

Consequently, for $t = 2$, we have

$$\begin{aligned} & z_2 = \rho z_1 + (1-\rho)\mu + \varepsilon_2 = \rho\mu + \rho\varepsilon_1 + (1-\rho)\mu + \varepsilon_2 \\ \Rightarrow \quad & z_2 = \mu + \rho\varepsilon_1 + \varepsilon_2 \end{aligned}$$

In fact, we can see a pattern (or more formally, use induction to write below). For any $t \in \{1, 2, \cdots\}$, we can write:

$$z_t = \mu + \sum_{j=1}^{t} \rho^{t-j} \varepsilon_j \tag{A}$$

The expression in (A) can be used as follows for the coding part. Let us define a vector V of length $N \geq 1$ whose t^{th} element, v_t, is such that $v_t = e_t$. That is, V is a vector of errors from periods 1 to N. To find z_t, then, we just need to find the value

$$\sum_{j=1}^{t} \rho^{t-j} v_j \tag{A2}$$

and add μ to this value. In `PyOpenCL`, we can calculate the expression in (A2) by using `GenericScanKernel`, as demonstrated similarly in Lab 2.

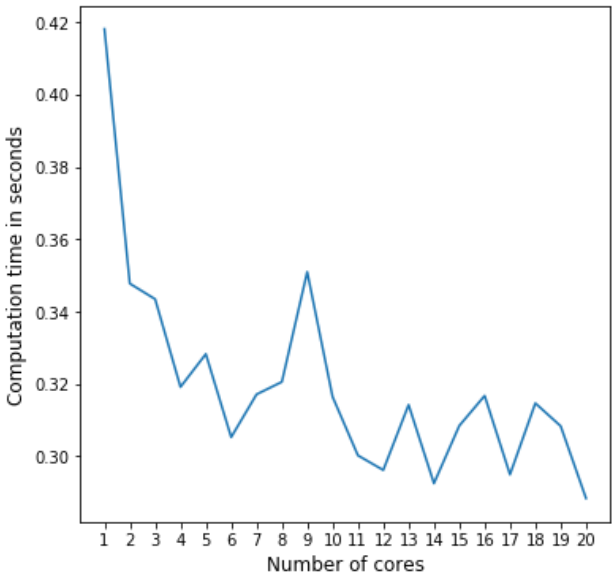
Question 1

The codes used for this question is given in the Coding Appendix's **Part A** (`q1_actual.py` and `q1.sbatch`).

1-(a) + 1-(b). I note that I used up to 20 cores (`ntasks=20`) as directed in the Piazza post. The computation time taken for each run is given in Figure 1.

As seen in Figure 1, the computation time actually decreases with the number of cores used, but is not linear as the question suggests. My reasoning for this happening is as follows. In my implementation, I used `numpy` operations, which allowed me to use a single for loop (over the time periods) instead of the nested for-loop suggested. Because `numpy` operations are quite fast for arrays of considerable sizes, I think there was not too heavy of a operation to begin with. I think that a part of the computation time might have actually

Figure 1: Computation Time per Cores Used



NOTE: The vertical lines indicate where the average-length-maximizing value of ρ is.

increased by different cores sending (and receiving) a sizeable array containing sub-simulation results; this would explain why computation time decreases non-linearly with the number of cores.

In the nested for-loop implementation, I think there would be no linear speedup as well due to communication between cores slowing down the process. In general, I think this shows that while parellelization is useful, some simpler tasks do not necessarily benefit massively from increased degree of parallelization.

Question 2

The codes used for this question is given in the Coding Appendix's **Part B** (`q2.py` and `q2.sbatch`).

2-(a).

The overall computation took approximately **0.251044** seconds.

2-(b).

In my `mpi4py` implementation, the fastest case was with 20 cores used; it took **0.288383** seconds approximately. Therefore, the PyOpenCL implementation is slightly faster than the 20-core `mpi4py` implementation. I believe that PyOpenCL implementation, leveraging on GPU computation, is better with matrix operations and therefore is able to provide a comparable computation speed to the 20-core implementation (which also uses `numpy` operations to speed things up as well).

Question 3

3-(a).

The codes used for question 3-(a) is given in the Coding Appendix's **Part C** (`q3.py` and `q3_a.sbatch`). I note that I used 20 cores (`ntasks=20`) as directed in the Piazza post. The overall computation took approximately **2.554788** seconds.

3-(b).

The codes used for question 3-(b) is given in the Coding Appendix's **Part D** (`q3_gpu.py` and `q3_gpu.sbatch`). The overall computation took approximately **2.418548** seconds.

While the PyOpenCL implementation of this part is slightly faster than the MPI for Python implementation, I would say that there is not much difference in computation time. I believe that this result is partly due to the I structured my codes. In the MPI for Python implementation, I was able to split the ρ values to be tested onto 20 cores in the Midway RCC. This parallelization saves time, but concurrently matrix operations (even with NumPy) are faster with PyOpenCL as it leverages GPU computing. However, in my PyOpenCL implementation, I was unable to compute for multiple values of ρ at the same time, either due to my lack of coding expertise or the way that `GenericScanKernel` is structured. The said implementation is still parallel in the sense that it quickly computes multiple lives for multiple periods at once, but I used linear programming to compute for individual values of ρ, which likely contributed to slower computation time.

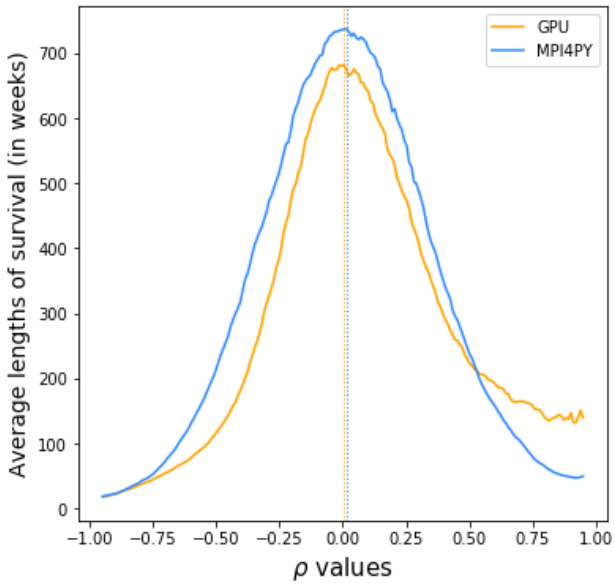
3-(c).

The requested plot is in Figure 2. While both cases have very similar average-length-maximizing ρ value, the distributions of average lengths are different. I think there could be a few reasons for why this is the case. Firstly, this could be due to the epsilon (ε) values, which are drawn differently in the two cases. Secondly, I think that there could be precision issues with PyOpenCL implementation as opposed to MPI for Python implementation. I was able to check that GPU computing through PyOpenCL (using `np.float32`) is not very precise with numbers of small absolute values; for instance, exponentiating 0.95 to the power of 4160 in regular NumPy application would be precise, but in PyOpenCL it would simply return 0. Therefore, by using PyOpenCL, I could have included more positive-health cases as nonpositive (including zero health) cases. This would explain why the graph for PyOpenCL implementation as a lower magnitude in general as well as being asymmetric.

3-(d).

The result is recorded in Table 1.

Figure 2: Values of ρ and Corresponding Averge Lengths Until Nonpositive Health



NOTE: The vertical lines indicate where the average-length-maximizing value of ρ is.

Table 1: Optimal ρ and Accompanied Average Length (Question 3)

Case	For Question	Optimal ρ	Average Length
MPI for Python	3-(a)	0.014322	737.468
PyOpenCL	3-(b)	0.004774	682.612

NOTE: Optimal ρ values were rounded to the nearest millionth. Average length is denominated in weeks.

Question 4

I note that I use `scipy.optimize.minimize_scalar` instead of `scipy.optimize.minimize`.

4-(a).

The codes used for question 4-(a) is given in the Coding Appendix's **Part E** (`q4.py` and `q4_try.sbatch`). I note that I used 20 cores (`ntasks=20`, so 50 simulations per core) as directed in the Piazza post. The overall computation took approximately **1.350500** seconds.

4-(b).

The codes used for question 4-(a) is given in the Coding Appendix's **Part F** (`q4_gpu.py` and `q4_gpu.sbatch`). The overall computation took approximately **1.347626** seconds.

4-(c).

The result is recorded in Table 2. Since we have broken down the entire 1000 simulations into 20 sub-parts for question 4-(a), I report the mean optimal ρ and standard deviation of the (sample) ρs. Similarly, I

also report the mean maximum average length and the average lengths' standard deviation. For question 4-(b), since the entire process is not broken down into different cores, I just report the optimized ρ and corresponding average length.

Table 2: Optimal ρ and Accompanied Average Length (Question 4)

Case	For Question	Optimal ρ	Average Length
MPI for Python	4-(a)	-0.003153	766.832
		(0.061507)	(80.984)
PyOpenCL	4-(b)	0.002540	682.466

NOTE: For MPI for Python, reported are mean optimal ρ values and average lengths over sub-simulations and respective standard deviations (in parentheses). Statistics related to the optimal ρ values were rounded to the nearest millionth, and those related to average lengths were rounds to the nearest thousandth. Average length is denominated in weeks.

4-(d).

The methods implemented in this question as opposed to the previous one are faster. I believe that this is due to the function `scipy.optimize.minimize_scalar` (or `scipy.optimize.minimize`) already being very much optimized for computation. Therefore, the grid search method – which may require some linear processes to be accompanied, due to issues like memory overflow – is slower. In fact, in my own codes, I have been able to remove the linear programming part for question 4-(b). For question 4-(a), the process was further parallelized by partitioning into sub-simulations.