COP 6526
Homework 2 Solution

September 29, 2025

Arman Sayan

---

1 Question 1: Parallel k-Means Clustering [50 points]

Dataset: MNIST digits (subset 10,000 images) (or sklearn.datasets.load_digits)

1. [12 points] Implement k-means clustering (sequential code). k=10.

> **Ans:**
> Please check the source code included in the .zip file named as
> **cop_6526_hw2_q1_sequential_arman_sayan.py**
> for the sequential implementation of k-means clustering.

2. [12 points] Parallelize both E-step (assignment) and M-step (update) using Python Multiprocessing. You may split dataset among processes. During the E-step (assignment), each process assigns points to nearest centers. During the M-step (update), each process handles a cluster by computing the mean of its assigned points.

> **Ans:**
> Please check the source code included in the .zip file named as
> **cop_6526_hw2_q1_multiprocessing_arman_sayan.py**
> for the parallel implementation of k-means clustering using Python Multiprocessing.

3. [14 points] Parallelize both E-step (assignment) and M-step (update) using mpi4py.

> **Ans:**
> Please check the source code included in the .zip file named as
> **cop_6526_hw2_q1_mpi_arman_sayan.py**
> for the parallel implementation of k-means clustering using mpi4py.

4. [12 points] Answer the following questions:
   (a) From your experiments, does increasing the number of processes in multiprocessing make the program run faster?
   (b) Does increasing the number of processes in mpi4py make the program run faster?
   (c) Does changing the number of processes have any effect on the accuracy of the K-means clustering results?"

Ans:

The runtime comparison of the three different implementations of k-Means clustering is summarized in Table 1. The experiments were conducted on a local machine with 6 physical CPU cores. Since multiprocessing does use logical CPU cores, which is 12 in this case, we tested up to 12 processes for the multiprocessing implementation. The accuracy of the clustering results was evaluated by comparing the predicted labels with the true labels of the dataset.

Method	# Processes	Runtime (s) ↓	Accuracy (%) ↑
`sequential`	1	**22.42**	46.80
`multiprocessing`	1	36.20	46.80
	2	23.91	46.80
	3	22.19	46.80
	4	20.66	46.80
	5	18.37	46.80
	6	20.19	46.80
	7	19.34	46.80
	8	18.62	46.80
	9	18.33	46.80
	10	19.35	46.80
	11	18.98	46.80
	12	**18.29**	46.80
`mpi4py`	1	35.22	47.40
	2	13.28	47.24
	3	7.57	49.58
	4	6.35	48.20
	5	9.28	42.60
	6	**5.93**	49.78

Table 1: Runtime and Accuracy of k-means clustering using different implementations and number of processes.

(a) The results show that increasing the number of processes in the multiprocessing implementation generally reduces runtime compared to using a single process, although the benefit is not linear. With one process, multiprocessing is slower than the sequential version due to process management overhead. However, as more processes are added, performance improves, and the runtime drops steadily, reaching its best value at 12 processes. After around 5–6 processes, the gains become smaller, and runtimes fluctuate slightly, reflecting overhead and diminishing returns. This indicates that multiprocessing can provide speedups, but its efficiency is limited by overhead and the shared-memory model.

(b) The MPI implementation demonstrates far stronger scaling behavior than multiprocessing. Runtime decreases sharply as the number of processes increases, falling from over 35 seconds with one process to under 6 seconds with six processes. This shows that MPI is highly effective at distributing the workload across processes. However, the results also reveal a non-linear trend like at five processes, the runtime increases compared to four, likely due to communication overhead or load imbalance. Overall, the results highlight that MPI achieves significant performance gains with more processes, but its efficiency can vary depending on how evenly the work and communication costs are distributed.

(c) Changing the number of processes does not significantly affect the accuracy of k-Means clustering. In the multiprocessing implementation, accuracy remains constant at 46.80% regardless of process count, showing that parallelism only impacts runtime. In the MPI implementation, accuracy fluctuates slightly between 42–50%, but these differences are due to the non-deterministic order of reductions, not the number of processes itself. We tried to use the same seed number for all runs to somehow control the random initialization of centroids. Therefore, we conclude that the number of processes influences execution time but has no significant impact on the clustering accuracy of the algorithm.

2 Question 2 [50 points]

Please parallelize the following nonlinear regression program using multiprocessing and MPI in python. According to the number of CPU cores in your computer (or one computer on stokes), you may use the same number of processes.

Here are the nonlinear regression program and data.

Hints:

(1) To parallelize the training, you can distribute the data to different processes at the beginning. Then under every epoch, every process calculates and summarizes gradients, and then sends them to the main process. Then, in the current epoch, the main process collects gradients from all processes, updates the model, and then propagates the model to every process. Here we do not consider mini-batch. Such kind of distributed/parallel machine learning is "synchronized", which produces the same result as sequential execution.

(2) Please use Numpy for efficiency.

1. [20 points] Multiprocessing in Python

Ans:

Please check the source code included in the .zip file named as

cop_6526_hw2_q2_multiprocessing_arman_sayan.py

for the parallel implementation of nonlinear regression program using Python Multiprocessing.

2. [20 points] MPI in Python

Ans:

Please check the source code included in the .zip file named as

cop_6526_hw2_q2_mpi_arman_sayan.py

for the parallel implementation of nonlinear regression program using MPI.

3. [10 points] Please compare the entire runtime of the three different implementations (sequential, multiprocessing, MPI) in a report.

Ans:

The runtime comparison of the three different implementations of the nonlinear regression program is summarized in Table 2. The experiments were conducted on a local machine with 6 physical CPU cores. Since multiprocessing does use logical CPU cores, which is 12 in this case, we tested up to 12 processes for the multiprocessing implementation. Runtime only includes the time taken for training the model, excluding data loading and preprocessing.

Method	**# Processes**	**Runtime (s) ↓**	
		Debug	No Debug
`sequential`	1	**-**	**46.13**
`multiprocessing`	1	**12.22**	11.04
	2	13.03	11.25
	3	12.89	**10.96**
	4	14.31	12.08
	5	15.64	13.24
	6	17.34	14.87
	7	19.95	18.94
	8	21.37	20.02
	9	24.20	21.23
	10	26.51	21.75
	11	28.50	22.56
	12	31.82	23.73
`mpi4py`	1	2.12	1.31
	2	1.71	0.91
	3	**1.62**	0.77
	4	1.76	0.74
	5	2.16	**0.64**
	6	2.34	0.65

Table 2: Runtime (with and without debug statements) of nonlinear regression using different implementations and number of processes.

The runtime results clearly highlight the trade-offs between the three implementations.

The sequential implementation serves as a baseline, completing in roughly 46 seconds, but it is unable to exploit modern multi-core systems.

The multiprocessing version achieves a moderate speedup at small core counts like up to about 3 processes, but quickly suffers from diminishing returns as the number of processes increases. This is largely due to Python's multiprocessing overhead, which outweigh the benefits of additional parallelism for a dataset of this size.

In contrast, the MPI implementation demonstrates superior scalability and efficiency. Even at a small number of processes, MPI outperforms both sequential and multiprocessing, due to its highly optimized collective operations and lower communication overhead. Interesting enough, MPI runtimes remain stable across process counts, with best performance observed around 4–6 processes.

Debug logging introduces measurable overhead in all methods, particularly in multiprocessing, where log output from many processes significantly slows execution.

Overall, these results show that while multiprocessing can provide modest gains over sequential execution, MPI is the best solution for efficient parallel training for our dataset and nonlinear regression program.