15618: Spring 2023 - Final Project Milestone Report

Wei-Lun Chiu (weilunc) (weilunc@andrew.cmu.edu)
Jhao-Ting Chen (jhaoting) (jhaoting@andrew.cmu.edu)

April 19, 2023

1 Task description

The objective of this project is to compare the performance and costs of executing data clustering algorithms on a multi-core machine with PyOMP and on a set of workers with comparable resource constraints of AWS instances with PySpark. The project will involve analyzing the serial, paralleled, and distributed implementation of two clustering algorithms: K-means and DBSCAN, to evaluate their trade-off between performance, cost, and different algorithmic strategies.

2 Updated Project Progress

Task	Not Start yet	In Progress	Complete
Serial K-means implementation			■
PyOMP K-means implementation			■
PyOMP K-means experiment			■
PySpark K-means implementation			■
PySpark K-means experiment			■
Conclusion for different K-means approaches		■	
Serial DBSCAN implementation			■
Multiprocessing DBSCAN implementation			■
Multiprocessing DBSCAN experiment			■
Conclusion for different DBSCAN approaches		■	
(optional) PySpark DBSCAN implementation	■		
(optional) K-means C++ implementation	■		
(optional) DBSCAN C++ implementation	■		
(Optional) Scale up K-means on large dataset	■		

3 Summarize of Project Progress

Our work involved parallelizing clustering algorithms, specifically K-means and DBSCAN, to improve their performance. We implemented different versions of these algorithms, including serial and parallel with PyOMP and PySpark for K-means and serial and parallel with multiprocessing for DBSCAN. To evaluate their efficiency, we conducted experiments on various hardware setups, such as personal computers, PSC, and AWS instances, and compared the parallelization techniques to the serial implementation. In addition, we provided details on the hardware setup used for the experiments, including the CPU and instance type, and the cost of the resources.

We've updated our progress on our final report. Link: https://jtchen0528.github.io/cmu-15618-project/assets/files/final_report.pdf

4 Goals, Deliverables, and Obstacles

Our step-by-step goals from project proposal are listed below. We mark the completed goals with ■, incomplete goal with □, and obstacles with ☒.

(a) ■ Implement k-means and DBSCAN serial algorithm in python, and parallelized version with PyOMP. Sample datasets would be MNIST [1], CIFAR-10 [2], and CIFAR-100.

(b) ■ Conduct experiments on GHC and PSC with different number of processors, analyze the speedups. Expecting speedups to be in proportion to the number of processors executed on.

(c) ☒ (Optional) Implement different k-means strategies such as grid-based, partition-based, or task-based for performance comparison and workload imbalances inspection.

(d) ☐ (Optional) Implement serial and parallelized program in C. Analyze the performance between OpenMP and PyOMP.

(e) ■ Implement k-means and DBSCAN distributed/parallelized algorithm with PySpark, on set of instances with similar hardware constraints added up as GHC/PSC.

(f) ■ Conduct experiments on performance difference, data communication overheads, and results between PySpark and PyOMP, with different number of workers.

(g) ☐ (Optional) Scale up the problem with larger datasets. Repeat the experiments.

(h) ■ Conclude the performance, costs, complexity trade-offs between renting/maintaining a supercomputer/cloud services.

Regarding the completed goals, we have successfully conducted all the required k-means experiments using MNIST dataset. In case larger datasets need to be processed, we may consider applying the K-means-$\|$[3]. However, for DBSCAN, we have identified that its performance in Python is not satisfactory, and therefore, we plan to develop another parallel implementation of DBSCAN using C++. At present, our focus on PySpark DBSCAN is relatively low.

Regarding the investigation of different K-means strategies in (c), we have observed that the partition-based data parallel model is particularly suitable for K-means. Investigating alternative strategies for K-means is deemed to be pointless.

5 Poster Session

Our demo will consist of a static poster presentation, which will be a visual display of the algorithm design, parallel approaches, and distributed approaches. We aim to provide a comprehensive overview of the different techniques employed in the parallelization of Kmeans and DBSCAN algorithms. In addition, we will conduct intensive analyses of the outcomes achieved through these various parallelism approaches.

6 Preliminary Results

We've updated our full progress on our final report. Link: https://jtchen0528.github.io/cmu-15618-project/assets/files/final_report.pdf. And here are some preliminary results.

6.1 K-means Multi-thread Parallelization with PyOMP

6.1.1 Serial Implementation Analysis

To ensure the correctness of our algorithm, we conducted preliminary experiments on clustered accuracy with number of clusters. For MNIST, which has 10 clusters, clustering data points into more clusters then inference the clusters into true labels yields better accuracy. The accuracy of our implementation is shown in Table 1. The centroids of k-means with $k = 10$ is shown in Fig 1.

The following experiments has the setup of $k = 10$ for k-means clustering.

Clusters(k)	64		36		10	
Implementation	Serial	PyOMP	Serial	PyOMP	Serial	PyOMP
Iterations	100	100	255	255	92	92
Accuracy	83.74%	83.74%	77.73%	77.73%	59.42%	59.42%
Homogeneity	77.62%	77.62%	71.40%	71.40%	50.88%	50.88%

Table 1: Correctness of serial and PyOMP implementation of K-means clustering

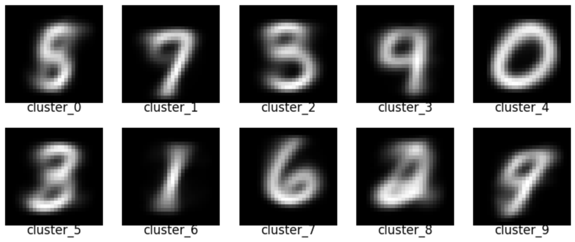


Figure 1: k-means clustered centroids with $k = 10$

6.1.2 Runtime decomposition with different nThreads

The runtime for the three stages in k-means are shown in Fig 2. As demonstrated in our experiments, Clustering Data stage accounts for most of the runtime. This is expected since clustering data points involves heavy computation on Euclidean distances. We leave our analysis of speedups in Sec 6.1.3.

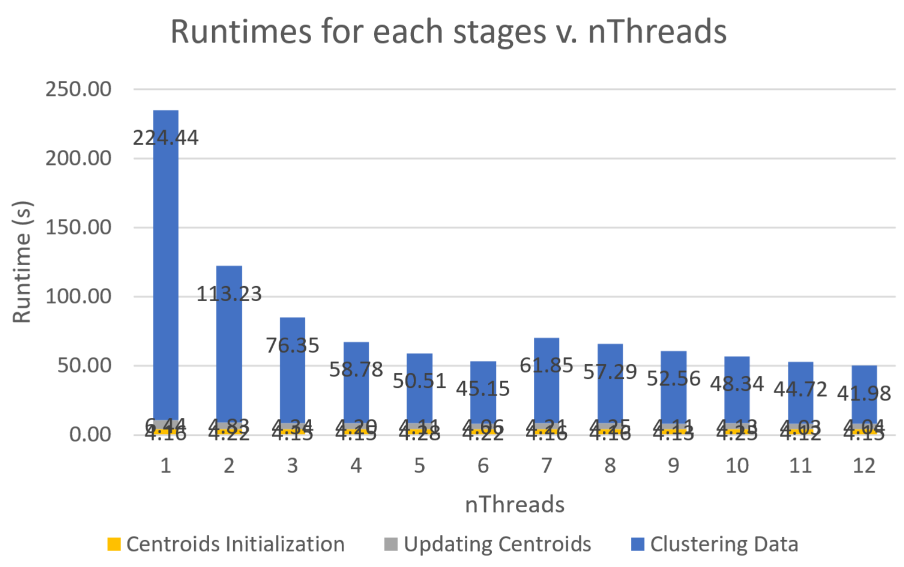
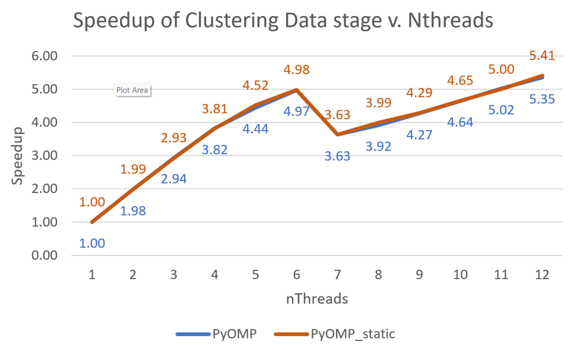


Figure 2: PyOMP K-means total runtime w.r.t. nThreads

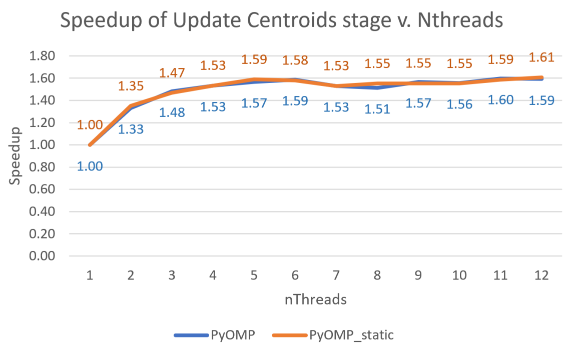
6.1.3 Clustering Data Analysis

Fig 3a illustrated the speedups of Clustering Data stage with respect to number of threads executing. From $nThread = 1$ to 6, the speedups increases almost linearly. An interesting phenomenon occurred when $nThread = 7$. The performance decreased but later increased with $nThread$ increased to 12. The reason may be the chip has only 6 cores, each with 2 threads. Synchronizing and communicating across the cores had less overheads. When adding threads that shares the resource in a core, the communication overheads and contentions are more apparent. Nevertheless, the speedup of 12 threads still out-performed the speedup of 6 threads.

We also experimented on the supported static scheduler. However, the result seems similar. Considering the fact that they have not implemented dynamic scheduler yet, we assume the implicit scheduler is the static scheduler.



(a) Clustering Data stage



(b) Updating Centroids stage

Figure 3: PyOMP K-means stage speedup w.r.t. nThreads

6.1.4 Atomic Implementation Analysis

Fig 3b illustrated the speedups of Updating Centroids stage with respect to number of threads executing. Observed in our experiment, the speedup were limited at around $1.6x$ when $nThread$ increases. The reason is that there were a necessary atomic action mentioned in **??**. The atomic operations in each threads contented for execution of writes to the summarized data points. Thus caused the speedup ceiling in the stage.

6.1.5 Different Optimization Approaches

We also implemented our serial algorithm with NumPy and the parallelized scikit-learn [4]. Fig 4 demonstrated the runtimes of PyOMP with 1 thread, PyOMP with 12 threads, pure serial implementation, NumPy optimized implementation, and scikit-learn implementation. We see that the same program but optimized with NumPy operations achieves quite a speedup comparing with plain implementation. However, our PyOMP implementation with 12 threads speedup still out-performed the NumPy-optimized program. Scikit-learn implements a more optimized K-means algorithm with built-in OpenMP support on multi-threading, which makes it the fastest. A more detailed analysis on CPU core utilization is shown in Sec 6.1.6.

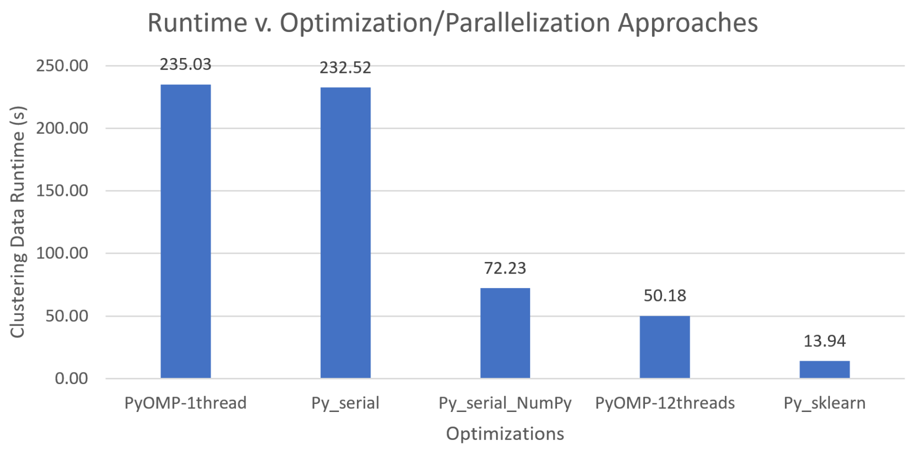
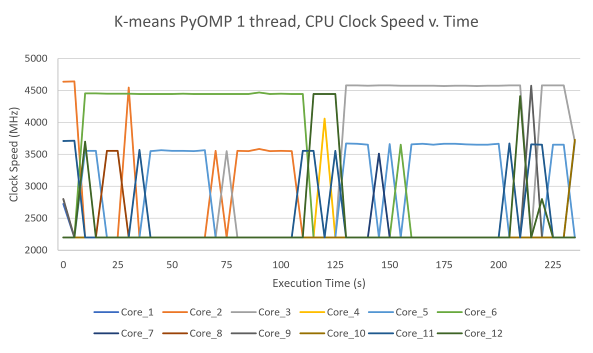


Figure 4: PyOMP K-means runtime with different optimization approaches

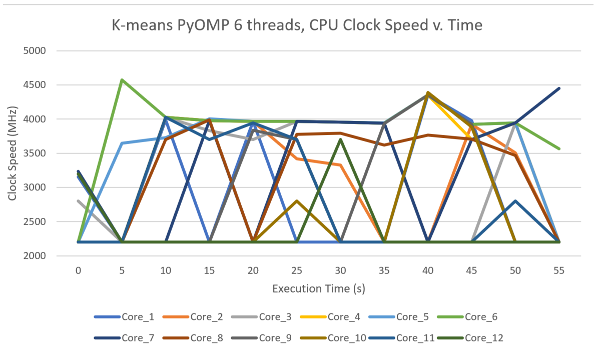
6.1.6 CPU Utilization Analysis

To better inspect the threads utilization during execution, we logged the CPU clock rate.

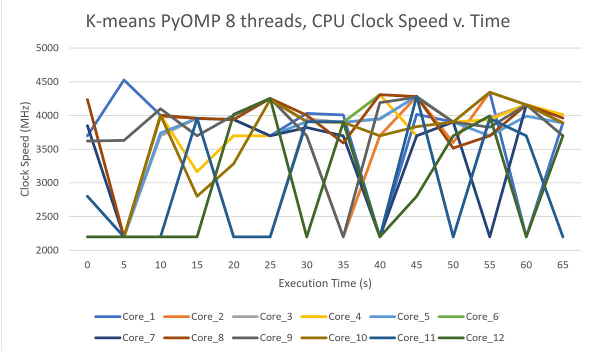
For PyOMP parallelized implementation, some logs are shown in Fig 5. As demonstrated in Fig 5a, there is always 1 core with the clock rate at about 4.5 GHz, close to the maximum clock speed of 4.6 GHz. The reason of threads changing is due to context switching between processors. As shown in Fig 5b and Fig 5c, with more threads running the program, the cores were utilized more heavily. Finally, Fig 5d demonstrated the cores were utilized in the same way, since every threads were executing exact same computation. One phenomenone worth noticing is the maximum clock speed differ between Fig 5a and 5d. We concluded two reasons for this: (i) communication overhead, and (ii) heterogeneous core design. The communication overhead between each threads may decrease the maximum clock speeds with more executing threads. The heterogeneous design of the CPU chip may benefit the utilization of single-thread process.



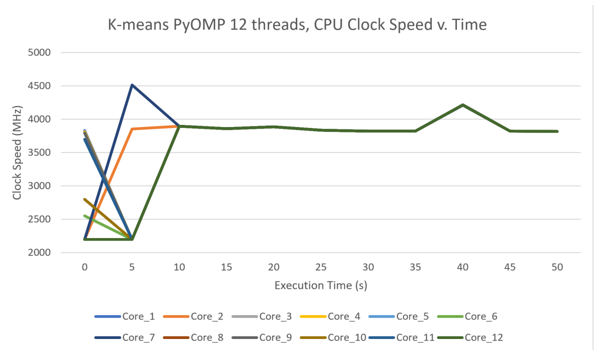
(a) PyOMP with 1 thread



(b) PyOMP with 6 threads



(c) PyOMP with 8 thread



(d) PyOMP with 12 threads

Figure 5: PyOMP K-means CPU Utilizations

6.1.7 Conclusion

As shown in the above analysis, our PyOMP parallelized k-means clustering achieved a 5.41x **speedup** with 6 cores (12 threads) comparing with the serial implementation. Our PyOMP parallelized implementation performed 44% better than NumPy-optimized version. Although it is difficult to surpass scikit-learn's efficient algorithm and parallelized execution, we consider our PyOMP implementation with 12 threads a success.

6.2 K-means Distributed Parallelization with PySpark

6.2.1 Implementation Correctness Analysis

To ensure the correctness of our algorithm, we tried to align every steps between PySpark and PyOMP. However, there are floating points error between AWS EC2 `t3.large` instance and AMD Ryzen5 5600x. Through experiments, we discovered that the error is about 10^{-9} for every data points each iteration. The slight difference caused about 5 in 60000 datapoints to be clustered in different cluster per iteration. Since the accuracy does not effect the performance analysis, we allowed it. The accuracy of PySpark k-means clustering is shown in Fig 6a and an example clusters shown in Fig 6b. During execution, we can see the execution time of each executor through PySpark server. A screenshot of a job executed on 12 executors is shonw in Fig 7.

Clusters (k)	10		
Implementation	Serial	PyOMP	PySpark
Iterations	92	92	100
Accuracy	59.42%	59.42%	64.62%
Homogeneity	50.88%	50.88%	55.93%

(a) Accuracy of PySpark K-means clustering

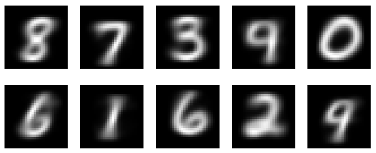

(b) K-means clustered centroids with $k = 10$

Figure 6: PySpark Accuracy Analysis

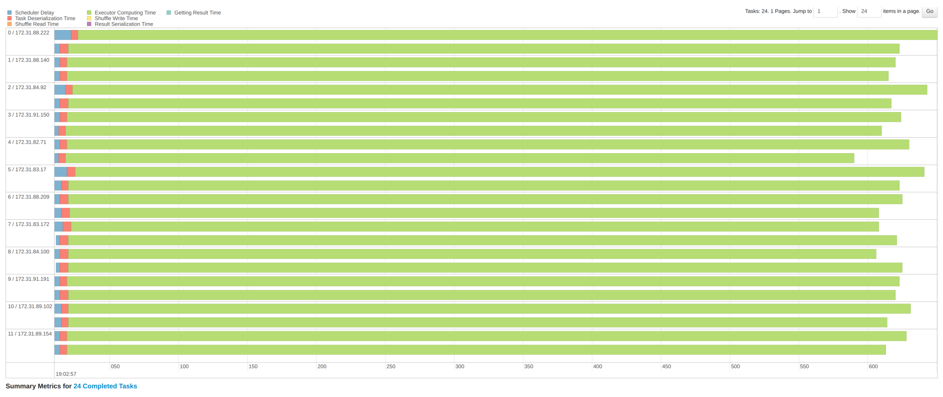


Figure 7: PySpark job executed on 12 executors

6.2.2 Runtime decomposition with different nThreads

We set up each experiments where $nThread$ is the number of `t3.large` nodes in a cluster. The dataset were partitioned into $nThread * 2$ since each node has 2 threads, each with half the clockd speed of AMD Ryzen5 5600x.

As we learned, Spark packs the several operations (stages) into a job. Jobs end with reduce tasks, such as `.collect()`, `.sum()`. Since Clustering Data stage is a map task which computes together with the later Updating Centroids stage, the two stages are analysis together.

The runtime for the two stages in k-means are shown in Fig 8. There is a drastic time difference between PySpark and PyOMP, where a 40 seconds runtime with 12 threads PyOMP turned to 24 minutes with 12 nodes PySpark.

Nonetheless, as demonstrated in the figure, the execution time of k-means with PySpark decrease linearly with number of executors. The same script executed on only 1 node takes about 4.2 hours, and 24 minutes on 12 nodes. The runtime includes execution, data synchronization, and Spark execution overhead, which is why the task took this long.

There is an anomaly when executing on 5 nodes, so we neglected the result. A detailed analysis is in Appendix **??**.

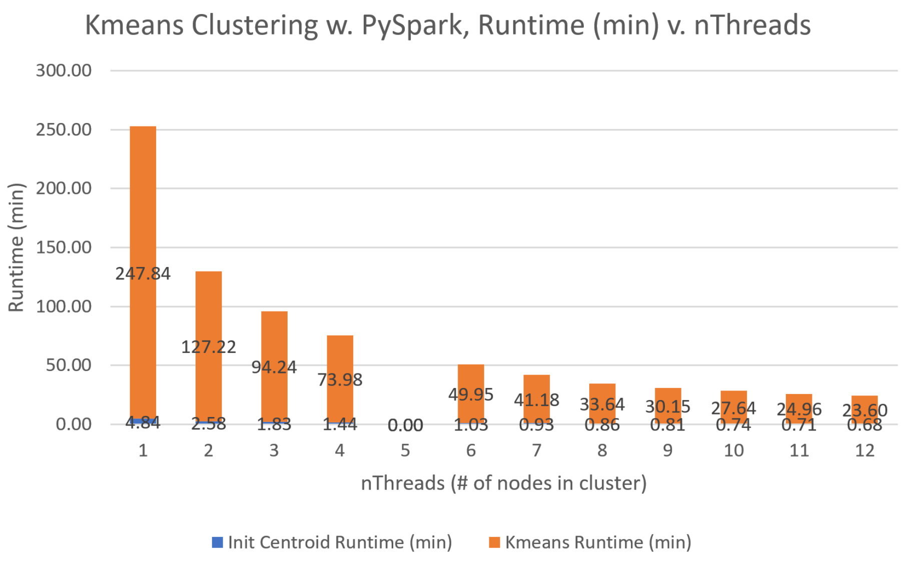
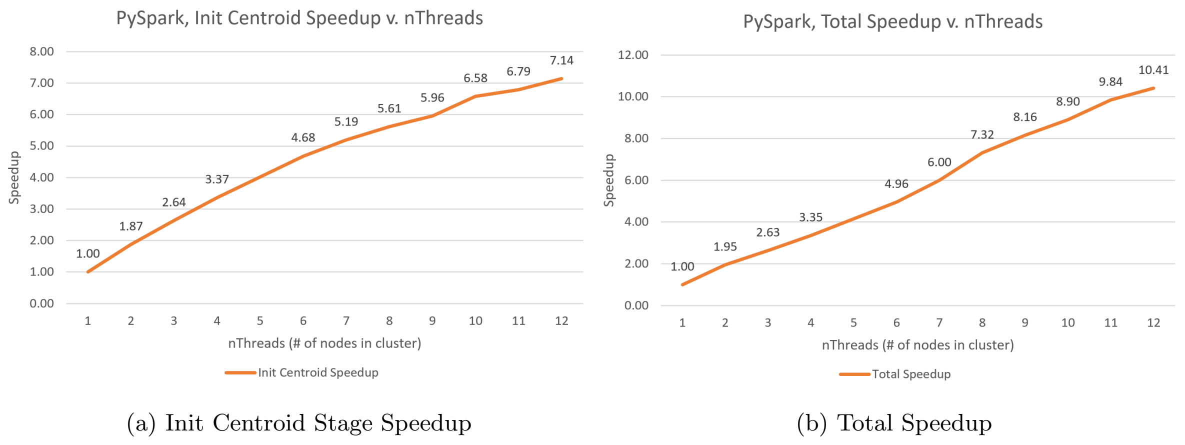


Figure 8: PySpark K-means total runtime w.r.t. nThreads

6.2.3 Speedup Analysis

In terms of speedup, Fig 9 illustrated the speedup of PySpark k-means implementation with respect to number of nodes in the cluster. The speedup between PySpark and PyOMP is quite similar from 1 thread to 6 threads. However, from 7 to 12 threads, the individual nodes do not suffer from issue mentioned in Sec 6.1.3. The speedup achieved 10.41x with 12 nodes executing at a time.



(a) Init Centroid Stage Speedup

(b) Total Speedup

Figure 9: PySpark K-means runtime w.r.t. nThreads

References

[1] Li Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[2] Alex Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[3] Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii, "Scalable k-means++," 2012.

[4] "scikit-learn/_kmeans.py," https://github.com/scikit-learn/scikit-learn/blob/9aaed4987/sklearn/cluster/_kmeans.py#L1161.