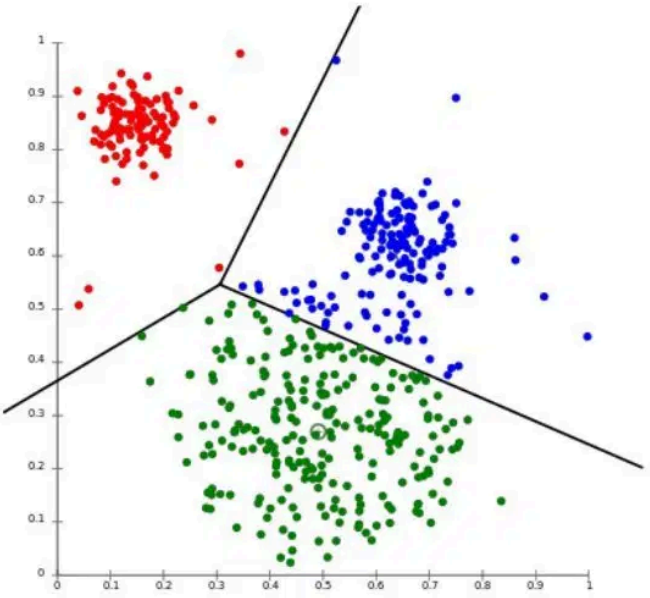
TECH REPORT:
K - MEANS



By Clara Alejos | Gonzalo Ortiz
Parallel Programming For Machine Learning

INDEX

1. INTRODUCTION

1.1 Objective

The main objective of this work is to analyze and compare the performance of the K-means clustering algorithm using two different approaches: sequential and parallel, leveraging OpenMP for parallelization. The sequential implementation processes data points iteratively, assigning them to clusters and updating centroids in a linear fashion. In contrast, the parallel implementation distributes these computations across multiple threads, reducing execution time by handling multiple data points simultaneously. The goal is to evaluate the advantages and limitations of parallelization in clustering tasks and measure performance improvements as the number of threads increases

1.2 Importance

K-means clustering is a widely used unsupervised learning algorithm applied in various domains. However, as the dataset size and dimensionality increase, the computational complexity of K-means also grows, making sequential implementations inefficient. Parallelization can significantly accelerate clustering, especially when handling large datasets, by distributing computations such as distance calculations and centroid updates across multiple processing cores. This study provides insights into the scalability of K-means clustering when implemented in parallel and highlights the impact of increasing the number of threads on execution time and clustering quality.

2. PROBLEM DESCRIPTION

2.1 K - Means Clustering

K-means is an unsupervised machine learning algorithm used to partition a dataset into k clusters. It iteratively assigns each data point to the nearest centroid and updates the centroids based on the mean of assigned points. This process continues until centroids stabilize or a maximum number of iterations is reached.

2.2 Data

To evaluate the performance of sequential and parallel implementations, we generate synthetic data points and test different values of k. The main parameters are:

- The dataset consists of 100,000 randomly generated points in a 2D space.
- The number of clusters k takes values from {3, 5, 10, 20, 50, 100}.
- The initial centroids are randomly selected from the dataset.
- A maximum of 100 iterations is allowed for convergence.
- The execution time is measured using different numbers of threads: 1, 2, 4, 8, and 16.

2.3 Challenges

- Managing the high computational cost due to repeated distance calculations and centroid updates.
- Efficient task division to balance workload in parallel execution.
- Evaluating the impact of different values of k on performance and convergence.
- Ensuring numerical stability and accuracy when increasing parallelization.

3. METHODOLOGY

3.1 Sequential Implementation

The sequential version of K-means follows a traditional iterative approach:

- **Assignment Step**: Each data point is assigned to the nearest centroid by computing the Euclidean distance.
- **Update Step**: The centroids are updated by computing the mean of the points assigned to each cluster.
- **Convergence Check**: If the centroids do not change significantly, the algorithm terminates early.
- **Time Measurement**: The execution time is measured tracking the start and end times.
- **Result Logging**: The execution times for different values of k are recorded in the output file.

Although this implementation is simple, it becomes computationally expensive as the dataset size and number of clusters increase due to the O(nk) complexity of each iteration.

3.3 Parallel Implementation

The parallel version leverages OpenMP to distribute computations across multiple threads, optimizing the most time-consuming parts of the algorithm. The key parallelized components are:

- **Parallel Distance Computation**: The assignment step is parallelized, allowing multiple threads to compute distances simultaneously.
- **Parallel Centroid Update**: Atomic operations are used to safely update centroid coordinates and count the number of points assigned to each cluster.
- **Synchronization Mechanism**: A shared variable ensures convergence checking is performed without race conditions.
- **Performance Scaling**: The algorithm is executed with different thread counts (2, 4, 8, 16) to analyze the impact of parallelization on speedup.

This approach efficiently distributes workload and reduces execution time, making K-means feasible for larger datasets.

3.4 Implementation Comparisons

The main differences between the sequential and parallel implementations are:

Aspect	**Sequential K-means**	**Parallel K-means**
Distance Calculation	Processed point by point	Computed in parallel for all points
Centroid Update	Sequential averaging	Parallel update with atomic operations
Work DIstribution	Single-threaded execution	Multi-threaded execution using OpenMP
Execution TIme	Increases linearly with data size	Scales efficiently with more threads

While the sequential version serves as a baseline, the parallel implementation significantly reduces computation time

3.5 Experimental Setup

- **Software:** Compiler GCC with support OpenMP
- **Libraries:** <iostream>, <fstream>, <vector>, <ctime>, <chrono>, <omp.h>, <cmath>, <cstdlib>

4. RESULTS

4.1 Performance Metrics

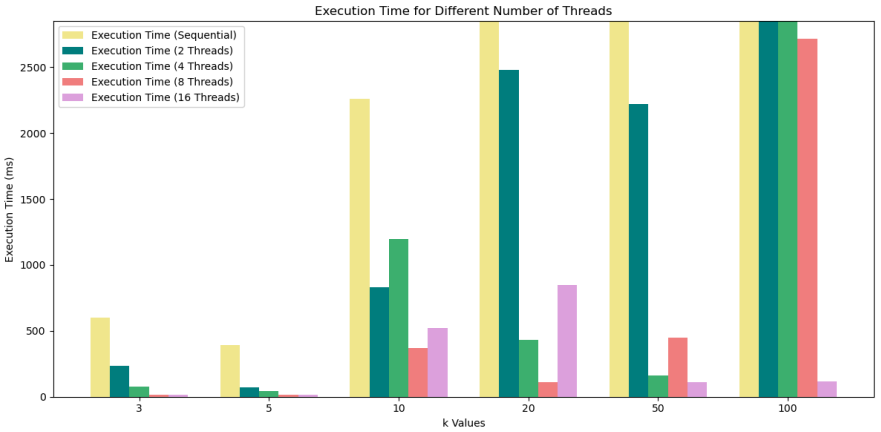
- **Execution Time:** Total time taken by the sequential and parallel implementations.
- **Speedup and Efficiency**

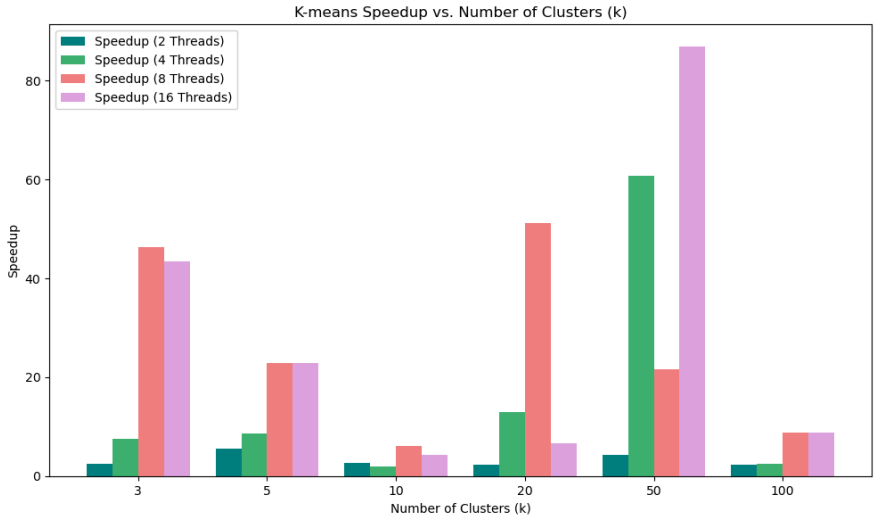
$$S = \frac{Tsequential}{Tparallel} \qquad E = \frac{S}{Num\ of\ threads}$$

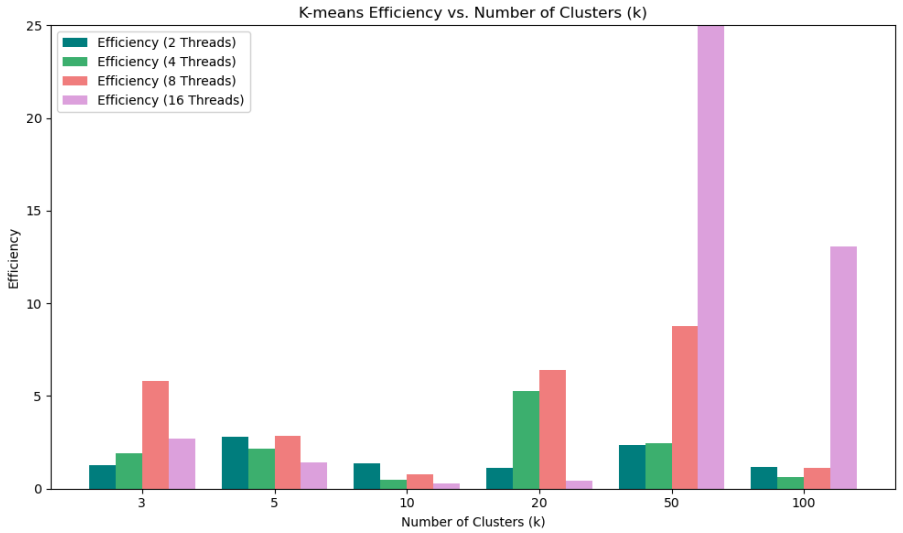
4.2 Observations

The sequential version was executed for only 5 passwords due to its high computational cost. It took 4557.402 seconds, so to estimate the time it would have taken to process 100 passwords, we assume the time grows linearly. By calculating the average time per password and multiplying by 100, we estimate this value.

4.3 Graphs


Execution Time for Different Number of Threads
Execution Time (Sequential)
Execution Time (2 Threads)
Execution Time (4 Threads)
Execution Time (8 Threads)
Execution Time (16 Threads)
Execution Time (ms)
0
500
1000
1500
2000
2500
3
5
10
20
50
100
k Values


K-means Speedup vs. Number of Clusters (k)
Speedup (2 Threads)
Speedup (4 Threads)
Speedup (8 Threads)
Speedup (16 Threads)
Speedup
0
20
40
60
80
3
5
10
20
50
100
Number of Clusters (k)


K-means Efficiency vs. Number of Clusters (k)
Efficiency (2 Threads)
Efficiency (4 Threads)
Efficiency (8 Threads)
Efficiency (16 Threads)
Efficiency
0
5
10
15
20
25
3
5
10
20
50
100
Number of Clusters (k)

5. DISCUSSION

5.1 Comparison Execution Time

The analysis of execution times for different thread configurations and values of k shows how parallelization impacts the performance of the K-means algorithm.

- **For k = 3:** The improvement is noticeable when using parallel threads. The sequential execution took 602 ms, while with 2 threads, the time decreased, and with 4 threads, it decreased even more. After 8 threads, the improvement becomes smaller, suggesting that after a certain point, adding more threads does not lead to significant improvement due to synchronization overhead.
- **For k = 5:** Similarly, the sequential execution time was 390 ms, and with 2 threads, it reduced significantly. With 4 threads, the time decreased a bit more, while with 8 and 16 threads, the time remained the same, indicating that parallelization improves performance, but efficiency stabilizes after a certain number of threads.
- **For k = 10:** As k increases, execution times grow considerably. Sequentially, the time was 2257 ms, and with 2 threads, it reduced to 829 ms. However, the performance with 4 threads was worse than with 2 threads, which could suggest a point where parallelization no longer improves performance due to resource overhead. With 8 threads, the time was 367 ms, and with 16 threads, it was worse than with 8, suggesting that in this case, more threads do not always improve performance.
- **For k = 20 and k = 50:** The pattern remains similar, with higher execution times as k increases. For example, for k=50k=50k=50, sequentially the time was 9669 ms, but with 2 threads, it reduced. However, the time increased with 8 threads compared to 4 threads, showing that parallelization doesn't always lead to a constant improvement.
- **For k = 100:** This value presents the biggest challenge in terms of performance. The sequential time was 23878 ms, and with 2 threads, it reduced to 10042 ms. Although the time with 4 threads was only slightly better than with 2 threads, with 8 threads, there was a great improvement, and with 16 threads, the time was 114 ms, showing a substantial improvement in execution with a higher number of threads.

In summary, parallelization has a positive impact on performance, but the benefits reduce as the number of threads increases due to synchronization overhead and resource exhaustion. Efficiency improvements become marginal after 8 threads in most cases.

5.2 Comparison SpeedUp And Efficiency

- **Speedup**: generally increases as more threads are used, but this increase tends to plateau at higher thread counts, especially for larger values of k.
- **Efficiency**: which is the ratio of speedup to the number of threads, shows a decreasing trend as the number of threads increases. This reflects the diminishing returns on performance gains with the use of more threads, especially when k is large.

5.3 Lessons Learned

The main takeaway is that while parallelization improves performance, the benefits decrease as more threads are added. Synchronization overhead and resource contention limit the performance gains. After a certain point, particularly with 8 threads or more, adding more threads does not lead to significant improvements. This is a classic example of diminishing returns in parallel processing.

Parallelizing the K-means algorithm improves performance for smaller values of k, with significant speedup and efficiency gains when using more threads. However, as k increases, the benefits diminish, and efficiency decreases due to overheads like synchronization and memory access delays. Beyond 8 threads, performance improvements are minimal, especially for larger values of k, as the problem becomes harder to parallelize efficiently. Therefore, for smaller k, parallelization is highly effective, but for larger k, the gains from additional threads are less noticeable.

6. CONCLUSIONS

6.1 Key Findings

- The parallel implementation significantly outperformed the sequential one, achieving a maximum speedup of 87x with 16 threads in the case of K-means with 50 clusters.
- Efficiency decreases with the number of threads due to synchronization overhead and resource contention. While speedup improves initially, efficiency drops with more threads.
- The optimal number of threads seems to be 8, providing a good balance between speed and efficiency for the given problem.
- Parallelization improves performance significantly for smaller values of k, but as k increases, the benefits of additional threads become less significant. For $k \leq 10$ parallelization is beneficial, particularly with 8 or more threads. For larger k, higher thread counts may not improve performance due to overheads.

6.2 Future Improvements

- **Better Load Balancing:** Implement dynamic work distribution among threads to improve efficiency, especially for highly parallelizable problems.
- **Vectorization:** Use SIMD instructions to optimize performance per thread, especially for mathematical operations such as calculating distances between points.

7. REFERENCES

- OpenMP: https://www.openmp.org/