

**PARALLEL AND DISTRIBUTED COMPUTING PROJECT REPORT**  
***PARALLELIZED DATA CLUSTERING ALGORITHM (Serial, OpenMP and MPI implementation)***

**GROUP MEMBERS:**Aimal Amir (21K3339)**,** Huzaifa Ahmed (21K3403),   
Jaleel Abbas (21K-3383).  
**SUBMISSION DATE:** 25th November,2024

**INTRODUCTION:**

**ABSTRACT:**

This report explores the parallelization of the K-Means clustering algorithm using three different approaches: Serial implementation, OpenMP for shared-memory parallelism, and MPI for distributed-memory systems. The objective is to evaluate the computational performance improvements and scalability achieved through parallelization while preserving clustering accuracy. Using a dataset of 10 million 3D points, experiments reveal significant reductions in execution time, demonstrating the advantages of parallelization for large-scale clustering tasks.

**GITHUB REPOSITORY:**

[***https://github.com/ronak197/Parallel-K-means-Clustering***](https://github.com/ronak197/Parallel-K-means-Clustering)

**BACKGROUND:**

K-Means clustering is a widely used unsupervised machine learning algorithm that partitions data into K clusters based on similarity. Despite its simplicity and efficiency, its computational cost increases significantly with larger datasets, making it challenging for real-time applications or resource-constrained environments.

**PROBLEM STATEMENT:**

The serial implementation of K-Means clustering is computationally intensive, especially for high-dimensional datasets or those with many data points. Parallelization provides an opportunity to reduce computational time by leveraging modern multi-core processors and distributed computing systems.

**OBJECTIVES:**

This project aims to:

1. Implement K-Means clustering in serial, OpenMP (shared memory), and MPI (distributed memory).
2. Compare the execution times and scalability of these implementations.
3. Analyze the trade-offs in parallelization, including speed-up, efficiency, and execution time.

**SCOPE:**

The implementations are tested on a dataset of 10 million 3D points. The programming is done in C with OpenMP and MPI, executed on systems supporting multi-threading and distributed memory.

**METHODOLOGY:**

**DATASET:**

A dataset of 10 million 3-dimensional points is used, stored in a file (points.txt) with each line containing the coordinates of a point. The dataset is designed to simulate real-world clustering challenges.

**IMPLEMENTATION:**

1. **Serial Implementation**:
   * Processes data point one at a time and computes distances sequentially.
   * Centroid updates occur in a single thread.
2. **OpenMP Implementation**:
   * Utilizes multi-core processors to parallelize key operations such as distance calculations and centroid updates.
   * Employs parallel loops with static scheduling for load balancing.
3. **MPI Implementation**:
   * Distributes data points among multiple processes using MPI\_Scatter.
   * Aggregates cluster assignments and centroids using MPI\_Reduce for global updates.
   * Scales to distributed systems by leveraging communication between processes.

**EVALUATION METRICS**

1. **Execution Time**:
   * Time taken for initialization and clustering iterations.
2. **Speed-Up**:
   * Ratio of execution time in the serial version to the parallel version.
3. **Efficiency**:
   * Emphasizes the balance between resource allocation and computational gain.

**FINDINGS:**

**OBSERVATION:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***Data Size*** | ***Threads/Processes*** | ***Execution Time*** | ***Speed-Up*** | ***Efficiency*** | ***Method*** |
| 10000000 | 1 | 1671 | 1.001795 | 1.001795 | OpenMP |
| 1000000 | 1 | 177.36 | 0.921628 | 0.921628 | OpenMP |
| 100000 | 1 | 18.1 | 0.946961 | 0.946961 | OpenMP |
| 10000 | 1 | 2.06 | 0.883495 | 0.883495 | OpenMP |
| 10000000 | 2 | 883.2 | 1.89538 | 0.94769 | OpenMP |
| 1000000 | 2 | 98.5 | 1.659492 | 0.829746 | OpenMP |
| 100000 | 2 | 9.61 | 1.783559 | 0.891779 | OpenMP |
| 10000 | 2 | 1.24 | 1.467742 | 0.733871 | OpenMP |
| 10000000 | 4 | 563.05 | 2.973093 | 0.743273 | OpenMP |
| 1000000 | 4 | 58.81 | 2.779459 | 0.694865 | OpenMP |
| 100000 | 4 | 5.78 | 2.965398 | 0.741349 | OpenMP |
| 10000 | 4 | 0.71 | 2.56338 | 0.640845 | OpenMP |
| 10000000 | 8 | 397.75 | 4.208674 | 0.526084 | OpenMP |
| 1000000 | 8 | 32.6 | 5.01411 | 0.626764 | OpenMP |
| 100000 | 8 | 3.63 | 4.721763 | 0.59022 | OpenMP |
| 10000 | 8 | 0.48 | 3.791667 | 0.473958 | OpenMP |
| 10000000 | 10 | 321.01 | 5.214791 | 0.521479 | OpenMP |
| 1000000 | 10 | 32.84 | 4.977467 | 0.497747 | OpenMP |
| 100000 | 10 | 3.79 | 4.522427 | 0.452243 | OpenMP |
| 10000 | 10 | 0.56 | 3.25 | 0.325 | OpenMP |
| 10000000 | 12 | 322.12 | 5.196821 | 0.433068 | OpenMP |
| 1000000 | 12 | 30.31 | 5.39294 | 0.449412 | OpenMP |
| 100000 | 12 | 3.44 | 4.982558 | 0.415213 | OpenMP |
| 10000 | 12 | 0.53 | 3.433962 | 0.286164 | OpenMP |
| 10000000 | 20 | 333.48 | 5.019791 | 0.25099 | OpenMP |
| 1000000 | 20 | 29.77 | 5.490763 | 0.274538 | OpenMP |
| 100000 | 20 | 3.22 | 5.322981 | 0.266149 | OpenMP |
| 10000 | 20 | 0.49 | 3.714286 | 0.185714 | OpenMP |
| 10000000 | 1 | 844 | 1.983412 | 1.983412 | MPI |
| 1000000 | 1 | 84.64 | 1.931238 | 1.931238 | MPI |
| 100000 | 1 | 8.41 | 2.03805 | 2.03805 | MPI |
| 10000 | 1 | 0.87 | 2.091954 | 2.091954 | MPI |
| 10000000 | 2 | 471.89 | 3.547437 | 1.773718 | MPI |
| 1000000 | 2 | 46.26 | 3.533506 | 1.766753 | MPI |
| 100000 | 2 | 5.24 | 3.270992 | 1.635496 | MPI |
| 10000 | 2 | 0.51 | 3.568627 | 1.784314 | MPI |
| 10000000 | 4 | 337 | 4.967359 | 1.24184 | MPI |
| 1000000 | 4 | 33.88 | 4.824675 | 1.206169 | MPI |
| 100000 | 4 | 3.61 | 4.747922 | 1.186981 | MPI |
| 10000 | 4 | 0.44 | 4.136364 | 1.034091 | MPI |
| 10000000 | 8 | 327.14 | 5.117075 | 0.639634 | MPI |
| 1000000 | 8 | 32.05 | 5.100156 | 0.63752 | MPI |
| 100000 | 8 | 4.04 | 4.242574 | 0.530322 | MPI |
| 10000 | 8 | 1.03 | 1.76699 | 0.220874 | MPI |
| 10000000 | 10 | 326.75 | 5.123183 | 0.512318 | MPI |
| 1000000 | 10 | 36.82 | 4.439435 | 0.443944 | MPI |
| 100000 | 10 | 4.87 | 3.519507 | 0.351951 | MPI |
| 10000 | 10 | 1.52 | 1.197368 | 0.119737 | MPI |
| 10000000 | 12 | 342.34 | 4.889876 | 0.40749 | MPI |
| 1000000 | 12 | 34.46 | 4.743471 | 0.395289 | MPI |
| 100000 | 12 | 6.18 | 2.773463 | 0.231122 | MPI |
| 10000 | 12 | 1.98 | 0.919192 | 0.076599 | MPI |
| 10000000 | 20 | 353.23 | 4.739122 | 0.236956 | MPI |
| 1000000 | 20 | 35.47 | 4.608401 | 0.23042 | MPI |
| 100000 | 20 | 8.05 | 2.129193 | 0.10646 | MPI |
| 10000 | 20 | 4.22 | 0.43128 | 0.021564 | MPI |

**ANALYSIS:  
A graph with a line

Description automatically generated**

1. **Serial Execution:**
   * Execution time increases linearly with the number of points.
   * The lack of parallelization results in the highest computation time.

**A graph of different colored lines

Description automatically generated**

**A graph with different colored lines

Description automatically generated**

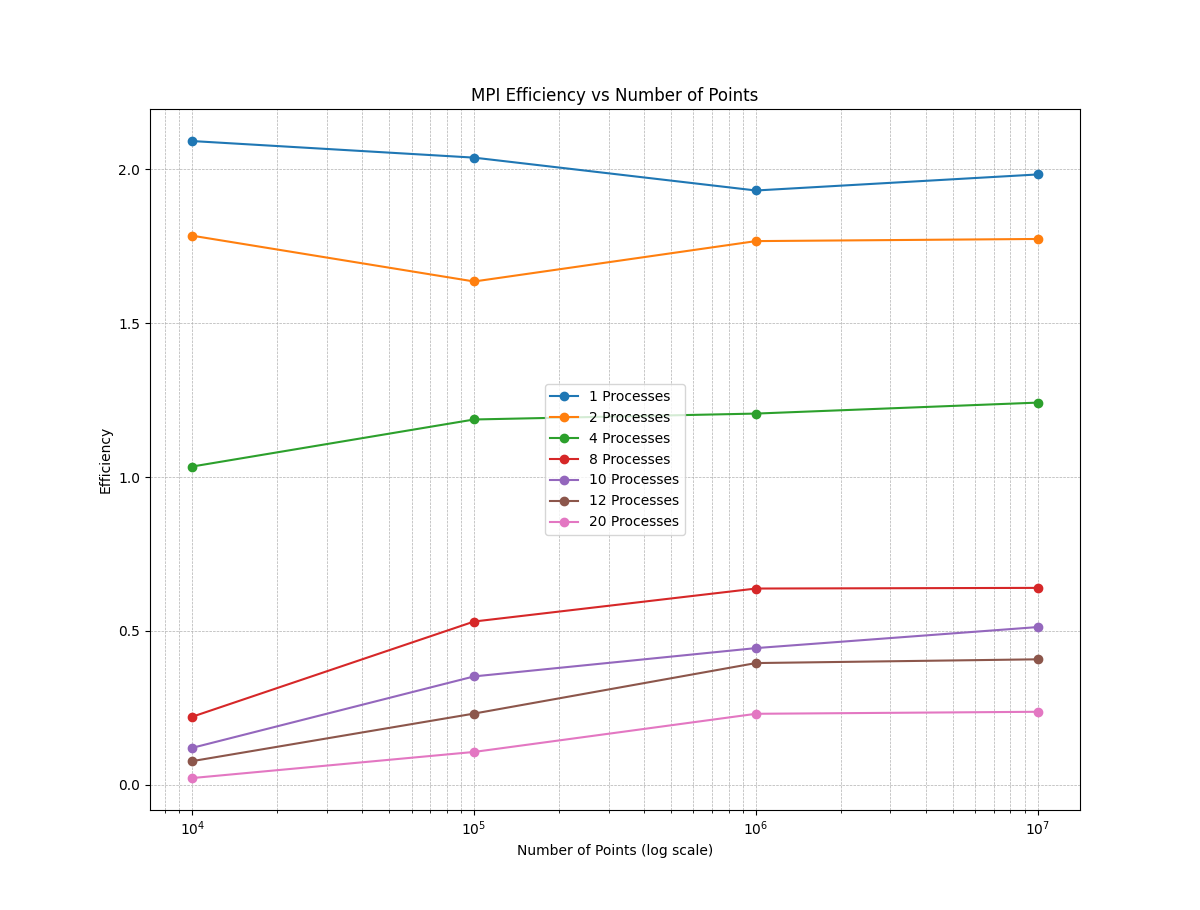
**A graph of different colored lines

Description automatically generated**

1. **OpenMP Performance:**
   * Execution time decreases significantly as the number of threads increases, particularly up to 8 threads.
   * Beyond 8 threads, the performance improvement diminishes due to factors like thread overhead and memory contention.
   * **Speed-Up:** OpenMP shows a consistent increase in speed-up with thread count, but efficiency drops as threads increase beyond 8.
   * **Efficiency:** OpenMP is most efficient at lower thread counts, with a significant drop at higher counts due to overhead.

**A graph of different colored lines

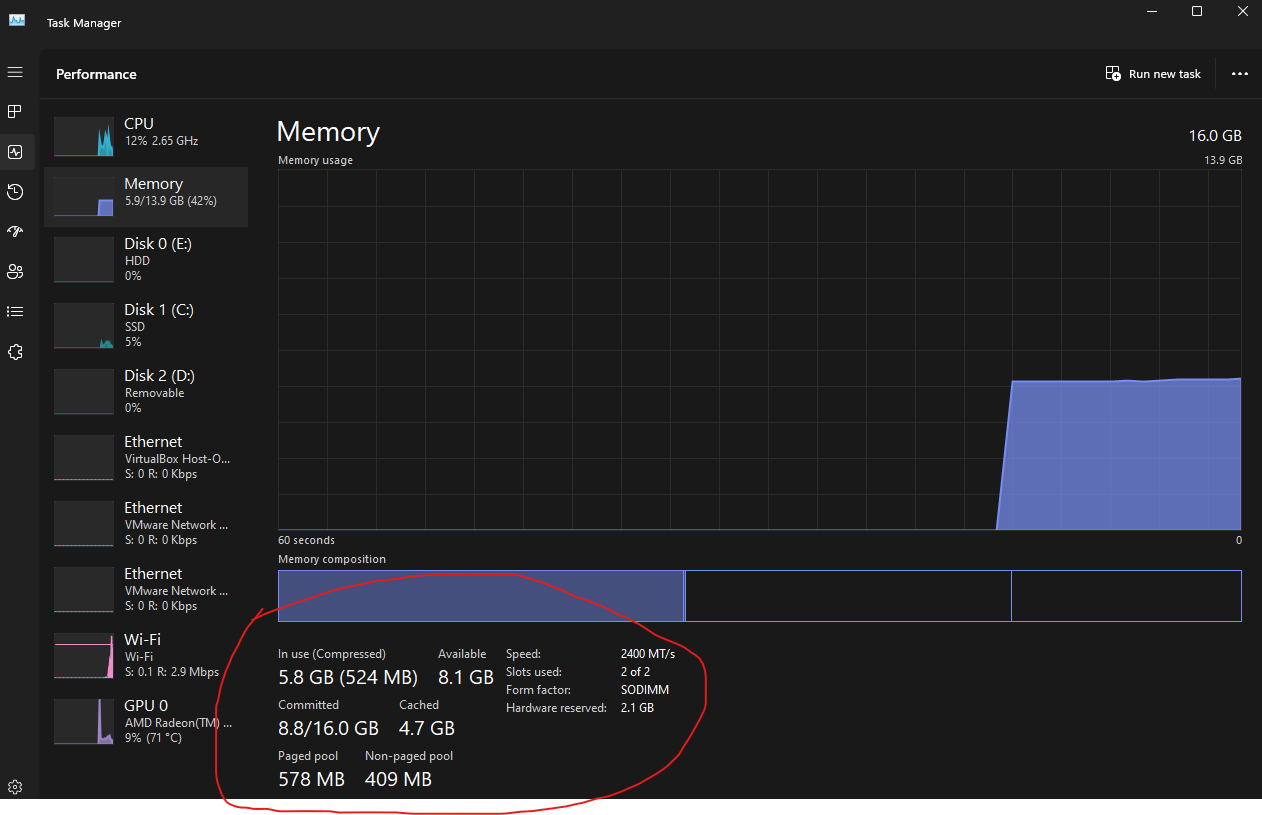
Description automatically generatedA graph with different colored lines

Description automatically generated  
**

1. **MPI Performance:**
   * Execution time decreases with the number of processes but plateaus after 4-8 processes.
   * Performance degrades slightly with higher process counts due to communication overhead.
   * **Speed-Up:** MPI achieves better speed-up for 4-8 processes but struggles with higher counts.
   * **Efficiency:** MPI achieves good efficiency up to 4 processes but faces diminishing returns and inefficiency due to communication bottlenecks.

**HARDWARE FOR OPENMP AND SERIAL: A screenshot of a computer

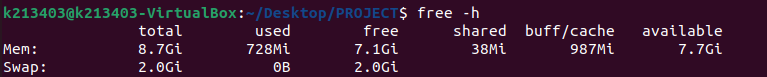
Description automatically generated**

****

**HARDWARE FOR MPI (windows has issues):**

A screenshot of a computer program

Description automatically generated



**CONCLUSION:**

**KEY FINDINGS:**

1. **Serial vs Parallel:**
   * Serial implementation is unsuitable for large datasets due to its high execution time.
   * Both OpenMP and MPI significantly reduce execution time compared to the serial implementation.
2. **OpenMP**:
   * Best suited for shared memory systems, achieving the highest efficiency and speed-up with 4-8 threads.
   * Limited scalability beyond 8 threads due to memory contention and thread management overhead.
3. **MPI**:
   * Effective for distributed memory systems, with better performance and scalability up to 8 processes.
   * Beyond 8 processes, communication overhead reduces efficiency, making it less practical for higher process counts.

**KEY TAKEAWAYS:**

1. **OpenMP**:
   * Use OpenMP for systems with shared memory (single-node setups).
   * Optimal thread count is close to the number of physical CPU cores (e.g., 4-8 threads).
2. **MPI**:
   * Use MPI for distributed systems where datasets are too large to fit into a single memory space.
   * Optimal process count for large datasets is 4-8, balancing computation and communication costs.