**Technical Report: Parallelization of K-Means Clustering with OpenMP**

**1. Introduction**

The goal of this project was to implement a parallelized version of the K-Means clustering algorithm using OpenMP to leverage multi-threaded computation and improve performance on large datasets. The implementation focuses on distributing computational tasks across multiple threads to achieve faster execution times, particularly for datasets with a significant number of points and clusters.

**2. Overview of the Algorithm**

K-Means is a clustering algorithm that partitions a set of points into a predefined number of clusters, k, by iteratively assigning points to the nearest centroid and updating the centroids based on the assigned points. The algorithm stops when the centroids converge or a predefined number of iterations is reached.

This implementation includes the following key steps:

1. **Initialization**: The centroids are initialized as the first k points in the dataset.
2. **Point Assignment**: Each point is assigned to the nearest centroid based on Euclidean distance.
3. **Centroid Update**: Centroids are recalculated as the center of gravity of the points assigned to them.
4. **Convergence Check**: The algorithm computes the Within-Cluster Sum of Squares (WCSS) to evaluate convergence.

**3. Parallelization Strategy**

The most computationally expensive step in K-Means is the assignment of points to the nearest centroid. This operation was parallelized using OpenMP as follows:

**3.1 Assignment Phase**

* Each thread operates on a subset of the points, determining the nearest centroid for its assigned points.
* Threads maintain private copies of cluster data to avoid race conditions.
* At the end of the entire process, a reduction step aggregates the private data into global structures.

**3.2 Centroid Update Phase**

* Centroid updates are parallelized using a loop over clusters, where each thread computes the new centroid coordinates for its assigned clusters (only if k>59).

**3.3 Convergence Check**

* The WCSS is calculated in parallel, with each thread contributing to a shared WCSS value using atomic operations or reductions to ensure correctness.

**4. Implementation Details**

The program is written in C++ with OpenMP directives for parallelization. Below is an overview of the key implementation details:

**4.1 Cluster Data Structure**

A custom Cluster struct is used to store centroids, cluster point sums, and WCSS values. Each thread operates on a private instance of this struct during the assignment phase.

**4.2 Parallel Regions**

OpenMP directives are used to create parallel regions:

* **Assignment Phase**: A #pragma omp parallel region with a private copy of the Cluster struct.
* **Reduction Step**: If in last iteration, a critical section combines private cluster data into the global cluster structure, otherwise another critical section combines the actual points number and X,Y coordinates sum for every Cluster.
* **Centroid Update Phase**: #pragma omp parallel for updates centroids for each cluster in parallel (if k>59).

**4.3 Dynamic Load Balancing**

To ensure balanced workloads across threads, every thread gets its own chunk of work by manually calculating its start index and end index.

**5. Performance Evaluation**

Performance was measured both using varying points number and a fixed number of clusters (k=3) and varying the number of Clusters and keeping a fixed points number (5.000.000). The program was run on a system with 6 CPU physical cores and hyperthreading, and the speedup was calculated as the mean of 5 attempts in both sequential and parallel versions.

| **Dataset Size (points)** | **Speedup** |
| --- | --- |
| 150,000 | 1.47x |
| 500,000 | 2.19x |
| 5,000,000 | 3.96x |
| 100,000,000 | 8.89x |
| 200,000,000 | 8.86x |
| 300,000,000 | 10.92x |
|  |  |
| **Dataset Size (clusters)** | **Speedup** |
| 3 | 4.56x |
| 5 | 5.58x |
| 9 | 9.01x |
| 12 | 9.63x |
| 15 | 8.62x |
| 25 | 6.21x |
|  |  |

**Observations**

* Speedup increases with dataset size, reflecting effective parallelization.
* Diminishing returns occur for extremely large datasets due to factors such as memory bandwidth.

**6. Compilation and Execution**

The program can be compiled and executed as follows:

**6.1 Compilation**

g++ -fopenmp -O3 kmeansFINAL.cpp -o kmeansFINAL

**6.2 Execution**

./kmeansFINAL

**7. Conclusion**

This project demonstrates the effective use of OpenMP to parallelize the K-Means algorithm. The results show significant speedups for large datasets, making the implementation suitable for high-performance clustering tasks. Future improvements could include:

1. Experimenting with alternative data partitioning strategies to further reduce synchronization overhead.
2. Profiling and optimizing memory access patterns to minimize cache misses.

**8. References**

* OpenMP Documentation: <https://www.openmp.org>