Parallel Image Segmentation using K-Means Clustering with MPI

Kisalay Ghosh

10th March 2025

Abstract

Image segmentation is a fundamental problem in image processing and computer vision. The **K-Means clustering** algorithm is widely used for this purpose but is computationally expensive when applied to large datasets. This report presents an **MPI-based parallel implementation** of the K-Means algorithm for image segmentation. Using **MPI_Scatter, MPI_Gather, and MPI_Reduce**, the workload is distributed among multiple processes to achieve better performance. The **execution time and speedup are analyzed** for different process counts to understand the scalability of the implementation. The results show that MPI-based parallelization significantly improves performance up to a certain limit, beyond which communication overhead starts affecting efficiency.

—

1 Introduction

The increasing size of image datasets and the need for real-time processing make **parallel computing** a necessity in image processing applications. **K-Means clustering** is an unsupervised learning algorithm used extensively in **image segmentation**, where pixels are grouped into clusters based on their color similarity. However, this method is **computationally expensive**, especially for high-resolution images.

To address this issue, we implement a **Message Passing Interface (MPI) based parallelization** of the K-Means algorithm. MPI enables us to distribute the workload across multiple processors, thereby reducing computation time. This report provides a **detailed explanation of the implementation, performance analysis, and speedup achieved using different numbers of processes**.

—

2 Algorithm and MPI Implementation

2.1 K-Means Clustering Overview

The K-Means algorithm follows these steps:

1. **Initialization**: Select k random pixels as initial centroids.
2. **Assignment Step**: Assign each pixel to the nearest centroid using Euclidean distance.
3. **Update Step**: Compute the new centroids by averaging the pixel values in each cluster.
4. **Iteration**: Repeat the process until convergence or a fixed number of iterations is reached.

2.2 Parallelization using MPI

To efficiently distribute the workload across multiple processes, we employ the **Master-Worker Model**:

- The **Master Process**:
  - Reads the image, extracts pixel data, and initializes centroids.
  - Uses **MPI_Scatter** to distribute pixel data to worker processes.
  - Collects updated cluster information from workers using **MPI_Reduce**.
  - Assigns final cluster values and writes the output image.
- The **Worker Processes**:
  - Receive image pixel data and assigned work.
  - Compute the closest cluster for each pixel.
  - Compute partial sums of cluster colors and send results back.

2.3 Mathematical Formulation

The Euclidean distance formula used for cluster assignment:

$$d = \sqrt{(R_i - R_j)^2 + (G_i - G_j)^2 + (B_i - B_j)^2} \tag{1}$$

The new centroid calculation formula:

$$\text{Centroid}_i = \frac{\sum \text{Pixel Colors}}{\text{Total Pixels in Cluster}} \tag{2}$$

—

3 Compilation and Execution

3.1 Compiling the Code

To compile the MPI-based K-Means clustering implementation, use the following command:

```
mpicc -o mpi_kmeans mpi_kmeans.c -lm
```

3.2 Running the Program

The program takes the following command-line arguments:

```
mpirun -np <num_processes> ./mpi_kmeans input.jpg output.jpg k num_iterations
```

For example, running the program with **4 processes, 5 clusters, and 10 iterations**:

```
mpirun -np 4 ./mpi_kmeans input.jpg output.jpg 5 10
```

If **more processes than available cores** are required, we use:

```
mpirun --oversubscribe -np 8 ./mpi_kmeans input.jpg output.jpg 5 10
```

—

4 Performance Analysis and Results

To analyze the performance, we measure **execution time** for different numbers of processes.

4.1 Execution Time vs Number of Processes

Observation:

- As the number of processes increases, execution time decreases initially.
- The lowest execution time is observed at **4 processes**, indicating optimal workload distribution.
- Beyond **4 processes**, execution time increases slightly due to **communication overhead**.
- The use of **8 processes shows inefficiencies** due to increased communication costs outweighing computation benefits.

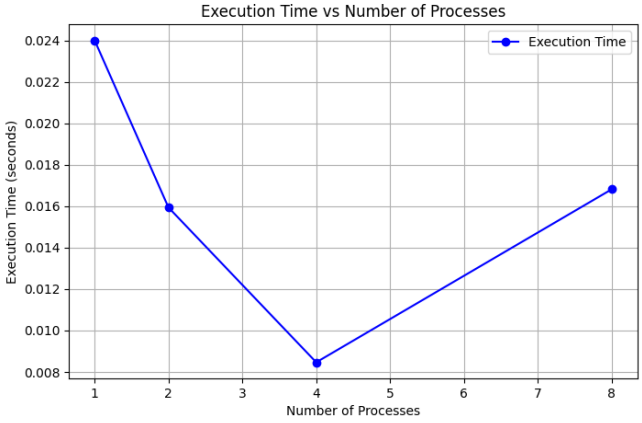

Figure 1: Execution Time vs Number of Processes

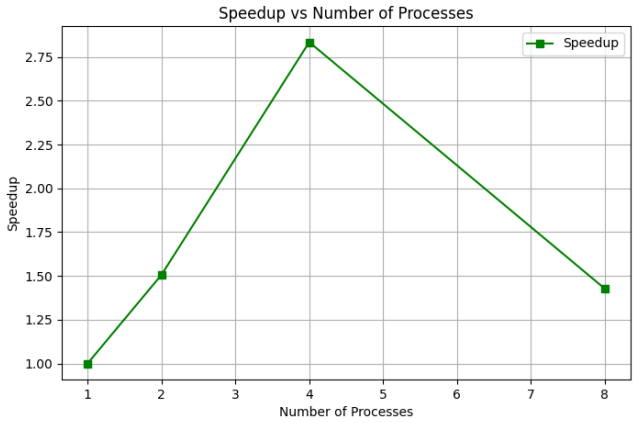

Figure 2: Speedup vs Number of Processes

4.2 Speedup Analysis

Observation:

- **Linear speedup** is observed up to **4 processes**, demonstrating efficient parallelization.
- **Diminishing returns** appear at **8 processes**, where increased communication overhead limits further speedup.
- **MPI scaling is effective**, but optimal performance depends on workload distribution and available hardware resources.

—

5 Discussion and Observations

- **Parallelization significantly reduces execution time**, making it suitable for large image datasets.
- **Speedup is close to ideal for 2 and 4 processes**, showing effective load balancing.
- **Beyond 4 processes, communication overhead increases**, reducing performance gains.
- **Dynamic workload balancing and hybrid MPI-OpenMP approaches** could further improve performance.

—

6 Conclusion

This project demonstrated a **parallel K-Means clustering implementation** using **MPI** for image segmentation. The experimental results indicate that MPI-based parallelization **effectively improves performance** up to **4 processes**, after which **communication overhead** starts impacting efficiency.

For future improvements:

- Implement **dynamic process allocation** to further optimize workload distribution.
- Use a **hybrid approach (MPI + OpenMP)** to leverage shared-memory parallelism.
- Apply **asynchronous communication techniques (MPI_Isend, MPI_Irecv)** to reduce communication bottlenecks.