# PC-2024/2025 Performance of Mean shift clustering using OpenMP

Edoardo Branchi
E-mail address
edoardo.branchi@edu.unifi.it

## Abstract

*This relation examines the performance of the Mean Shift clustering algorithm implemented using two approaches: a sequential method and a parallelized version with OpenMP. Mean Shift is a computationally intensive algorithm due to its iterative nature so by utilizing OpenMP we should see increased performance. The analysis compares execution time adn speedup, discussing also the difficulties of the parallel implementation. Results demonstrate that OpenMP achieves significant performance gains on multi-core processors while introducing a little bit of parallelization complexity.*

## 1. Introduction

Mean Shift Clustering is an algorithm used for data clustering and image segmentation. It identifies clusters by iteratively shifting data points towards the densest regions in feature space, guided by a kernel density estimation.

### 1.1. Key Characteristics

- **Non-Parametric:** Mean Shift does not require specifying the number of clusters before execution. The algorithm automatically identifies clusters.
- **Density-Based:** the goal of Mean Shift is to locate the maximum of the underlying density function.
- **Kernel Function:** The algorithm uses a kernel to weigh nearby points, giving higher importance to points closer to the current estimate.

### 1.2. General implementation

- **initialization:** Each data point is treated as a candidate cluster center.
- **Iterative Shifting:** For each data point:
  1. Define a search window (radius or bandwidth).
  2. Calculate the weighted mean of all points within the window.
  3. Shift the data point towards this mean.
  4. Repeat until the shift is smaller than a threshold (convergence).
- **Cluster Formation:** After convergence, points that converge to the same location are grouped into a single cluster.

## 2. My Implemetation Overview

My implementation is designed specifically for images. I clusters pixels based solely on color(LAB space) without considering spatial positions of the pixels, in other implementations sometimes pixels are clusterized also by position.

For the kernel i decided to use an ”hard” threshold to calculate weights, in particular if the Euclidean distance between points is less than the radius, I set weight = 1, otherwise, weight = 0. In this way all the pixel in the radius are treated the same way. In other implementation they usually use a Gaussian kernel or similar, where the weight decreases as the point is more far away in respect to the center.

When the computation is done I output using CV2 an image where the pixel colors are shifted to the cluster centers.

### 2.1. Sequential Mean Shift

In the sequential version the image is processed pixel by pixel during the Iterative Shifitng phase, this will cause a performance bottleneck as the Euclidean distance (also the weight but not in this case), are calculated in a sequential order.

### 2.2. Parallelized Mean Shift (OpenMP)

In the parallelized version, as each pixel calculation is independent from the other we can use the openMP directive:

```
#pragma omp parallel for
```

to share the computational across all available threads. In this way we should see a significant performance increase without adding practically no complications to the code.

## 3. Experimental Setup

The tests are conducted on a test setup with the following components. The processor has 4 cores and 8 threads .

| | |
|---|---|
| Nome dispositivo | edoardo-ubuntu 〉 |
| Memoria | 15,5 GiB |
| Processore | Intel® Core™ i7-3610QM CPU @ 2.30GHz × 8 |
| Grafica | NVC1 / Intel® HD Graphics 4000 (IVB GT2) |
| Capacità disco | 480,1 GB |
| Nome SO | Ubuntu 20.04.6 LTS |
| Tipo SO | 64-bit |

Figure 1. Specs of the test environment

The algorithm is applied to three images of various dimensions, for simplicity I will refer to it as

- **Small** image: 620x320 pixels.
- **Medium** image: 1280x853 pixels.
- **High** image: 2600x1498 pixels.

An example can be seen in Figure 2.

Figure 2. One of the images used for testing

To produce results the execution time is measured only on the computational part of the program leaving out the load and the display of the result and the color space conversions using ”chrono::high_resolution_clock”.

[1]https://www.intel.com/content/www/us/en/products/sku/64899/intel-core-i73610qm-processor-6m-cache-up-to-3-30-ghz/specifications.html

## 4. Results and Discussion

The plot showed in **Figure 3** represent data where each time measure is taken 5 times for each different radius size, on the same **Small** image then an average is computed as shown in the following sections. The one showed in **Figure 4** measure also 5 times and compute an average but fixing the radius size and varying the image sizes.

### 4.1. Performance Comparison

The execution time for the sequential implementation (blue line) increases significantly as the pixel radius increases, on the other hand, the parallel one using OpenMP (orange line) shows a much slower increase in execution time and consistently outperforming the sequential approach. The difference in steepness between the two indicate that the more the load is intensive the more the runtime benefits from parallelization.

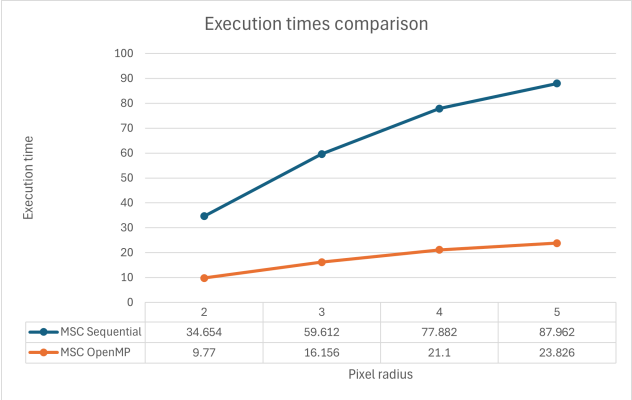


Figure 3. Execution time difference in sequential and parallel execution

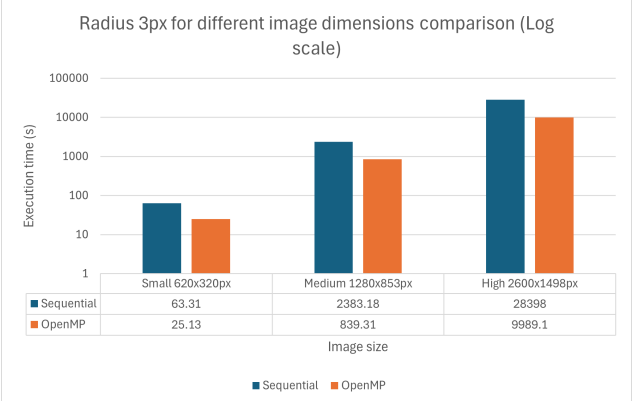


Figure 4. Execution time with fixed radius and varying image sizes

In Figure 4 we can see that the execution times increase exponentially both on the sequential approach

and the OpenMP approach, but the latter one constantly outperforms the first one by a factor of at least 2x. EWSSSSSSSSSSSSSSSSSSSSSSSSSSS In the following figures (Figure 5 and 6) we can see how the workload is shared among all available threads meaning the OpenMP directive is indeed working as expected.

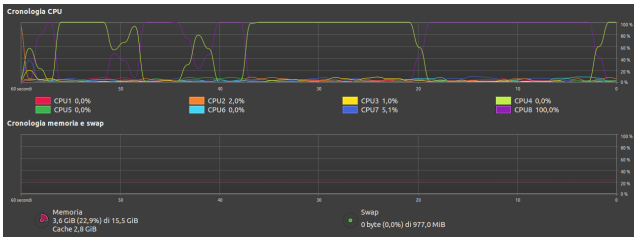


Figure 5. Cpu usage during sequential execution


Figure 6. Cpu usage during parallel execution

## 5. Conclusion

Between the two different implementation we obtained a **speedup of ≈ 3.5x** this only by adding a single line of code, so the tradeoff between complexity and performance gains is extremely favorable. Must be noted that while this might be a simplified example it still shows the capabilities and the advantages of thinking about a problem using the parallel point of view.

## 6. GitHub repository

Mean Shift Clustering
Mean Shift Clustering OpenMp