Meanshift Algorithm

Claudio Pagnini
E-mail address
claudio.pagnini@stud.unifi.it

Abstract

Mean shift clustering is a popular unsupervised learning algorithm used to identify clusters of data points in a high-dimensional feature space. In this work, we present an implementation of mean shift clustering using C++ and OpenMP to achieve parallel processing. Our implementation uses a distance metric to determine the similarity between data points and iteratively shifts each point towards the direction of maximum density until convergence. We assess the efficiency of our implementation on various 2D point datasets and illustrate the impact of the number of threads and maximum iterations on the clustering outcomes. Additionally, we compare the results obtained using two different data structure designs: Array of Structures (AoS) and Structure of Arrays (SoA).

1 Future Distribution Permission

The author of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

2 Introduction

Mean shift is a non-parametric clustering algorithm that aims to find the modes of a probability density function. It has been successfully used in various applications such as image segmentation, object tracking, and computer vision. In this project, we have implemented the meanshift algorithm in C++ and parallelized it using OpenMP. The goal of this project is to investigate the speedup achieved by parallelizing the meanshift algorithm and to compare it with the serial implementation.

The meanshift algorithm works by iteratively shifting the data points towards the direction of the maximum increase in the density. The shift is determined by the mean of the data points that lie within a certain radius, known as the kernel bandwidth. The position of the kernel is then updated to the new mean, and the process is repeated until convergence. The algorithm can be summarized in the following steps:

1. Initialize the positions of the kernels at the data points.
2. For each kernel, compute the mean shift vector, which is the direction of the maximum increase in the density.
3. Update the position of the kernel by shifting it towards the mean shift vector.
4. Repeat steps 2-3 until convergence.

There are several implementation of kernel function. In our implementation we have chosen to use (1) where λ represents the radius of the region of interest of which points are used for the shifting.

$$K(x) = \begin{cases} 1 & \text{if } x \geq \lambda \\ 0 & \text{if } x > \lambda \end{cases} \tag{1}$$

$$K(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{2}$$

2.1 AoS vs SoA design structure

In the context of computer science and software engineering, "AoS" and "SoA" stand for "Array of Structures" and "Structure of Arrays," respectively. They are two different ways of organizing data in memory, particularly when dealing with collections of related elements. AoS (Array of Structures) groups related data into an array of structures, while SoA (Structure of Arrays) organizes data into separate arrays based on attributes. AoS is intuitive and easy to use, while SoA can offer performance benefits in certain scenarios, especially for large datasets and optimized memory access.

3 Proposed Approach

In this paper we propose 2 different implementations of the algorithm:

1. A sequential version with Cpp
2. A parallel version with OpenMP using AoS and SoA architectures

It's been choosen to build implementations of Meanshift that work in 2-dimensional Euclidean spaces using the Euclidean distance to measure the distance between two points. The implementation is based on two array: the original array of points (*points*) and the shifted array of points (*new_points*).

3.1 Sequential Cpp

The implementation with Cpp just follow the principle of Meanshift. The following two pseudocodes shows the main and the sub-routine used to shift the points

Alg. 1 Meanshift **Main**

```
new_points<-original_points
REPEAT until MAX_ITERATION
    MeanShift(original_points,new_points, lambda)
```

Alg. 2 Meanshift **Sub-routine**

```
weight<-0
REPEAT for each points
    REPEAT until point_size
        distance<-Euclidean_Distance(point)
        if(distance<lambda)
            new_point <- new_point + original_point
            weight <- weight +1
    if(weight!=0)
        new_point <- new_point / weight
    else
        new_point<-point
```

3.2 OpenMp

The OpenMP API allow to transform the sequential version into a parallel one with several directives. It takes the same sequential implementation with minimal changes called *pragma omp parallel* directive shown in **Alg 4.** . As same as the sequential solution, there is a main which calls in order **Alg 3.** and then **Alg 4.**. Unlike the sequential implementation, every points assigned to each cluster may risk to be executed by one ore more threads at time. So, for this reason, we need to call that section "critical". The *firstprivate(points_ size)* clause is used to make a private copy of the points_size variable for each thread. By using *firstprivate*, each thread will have its own local copy of points_size initialized with the value from the outer scope. The *schedule(static)* clause specifies a static loop scheduling, where iterations are divided into chunks of equal size and assigned to threads in a round-robin fashion.

Alg. 3 Meanshift **Main**

```
new_points<-original_points
REPEAT until MAX_ITERATION
    MeanShift(original_points,new_points, lambda)
```

Alg. 4 Meanshift **Sub-routine**

```
weight<—0
#pragma omp parallel firstprivate(points_size)
#pragma  omp for schedule(static)
REPEAT for each points
    REPEAT until point_size
        distance<—Euclidean_Distance(point)
        if(distance<lambda)
            new_point <— new_point + original_point
            weight <— weight +1
    if(weight!=0)
        new_point <— new_point / weight
    else
        new_point<—point
```

Alg. 5 Meanshift **Non-zero shifted points**

```
REPEAT for each points
         if(shifted_points !=0)
             non—zeroPoints<—point
```

4 Experimental Results

The metric used to compare the performances of the sequential algorithm with the OpenMP the speedup, computed as:

$$S = \frac{t_S}{t_P}$$

where t_S and t_P are respectively the execution time of the sequential and the parallel implementation. The datasets used to evaluate the different implementations have been generated with the make_blob() function of sklearn.datasets [1]. They are gaussian distributions with 5 centers and standard deviation equal to 0.5 and are composed by respectively 100, 1000, 10000 and 100000 2D points.

The tests have been executed on a Laptdop that do not allow logical cores with:

- OS: Windows 11 Pro
- CPU: AMD Ryzen 5 4500U CPU @2.38GHz

Parameter used for each experiment has been:

- $\lambda = 1$
- MAXITERATION = 10

In order to obtain statistically significant data, each iteration has been performed in experimental triplicate. The statistical mean of the three results was then calculated and reported as final results.

4.1 OpenMP

To evaluate the performances of the OpenMP implementation, it has been executed on each dataset with the same number of threads comparing AoS vs SoA architectures. The results for the 100, 1000, 10000 and 100000 dataset are shown respectively in Table 1 and 2.

Dim	Seq Time	AoS Time	Speedup
100	0.0016 s	0.0021	0.761
1000	1.178 s	0.473 s	2.490
10000	112.966s	40.072 s	2.81
100000	11175.633 s	4031.578 s	2.77

Table 1. Speedup for 100, 1000, 10000, 100000 points using AoS arhchitecture

Dim	Seq Time	SoA Time	Speedup
100	0.0016 s	0.0017	0.941
1000	1.178 s	0.3990 s	2.952
10000	112.966s	35.188 s	3.210
100000	11175.633 s	3467.315 s	3.223

Table 2. Speedup for 100, 1000, 10000, 100000 points using AoS arhchitecture

For the dataset containing 100 points (Table 1 and 2), it is intriguing to observe that the speedup is below 1, indicating that the sequential version performs better than the parallel versions. This is attributed to the fact that parallelization introduces overhead, such as thread creation, synchronization, and communication between threads, and if the workload is not substantial enough, this overhead can outweigh the advantages of parallel execution. As expected, this phenomenon disappears in datasets with more points, and using more threads reduces the overall execution time.

4.2 Comparison

After conducting a comprehensive analysis, the best results for each dataset dimension and implementation have been compared. Table 3 presents the findings, demonstrating that the OpenMP SoA algorithm outperforms both the sequential and OpenMP AoS implementations, with the exception of the dataset containing 100 points, where the sequential approach performs better.

Dim	Sequential	OpenMP AoS	AoS SpeedUp	OpenMP SoA	SoA Speedup
100	0.0016 s	0.0021	0.761	0.0017	**0.941**
1000	1.178 s	0.473 s	2.490	0.3990 s	**2.952**
10000	112.966s	40.072 s	2.81	35.188 s	**3.210**
100000	11175.633 s	4031.578 s	2.77	3467.315 s	**3.223**

Table 3. Global comparison between sequential, OpenMP AoS and OpenMP SoA. Best results in bold

5 Conclusions

This work focuses on the Meanshift clustering algorithm and explores its potential for parallel computing. By leveraging its inherently embarassingly parallel structure, we developed a parallel implementation using OpenMP. With a simple directive, the OpenMP version achieved a notable speedup of approximately 3.2, making the algorithm significantly faster than its sequential counterpart.

Furthermore, we experimented with two versions of OpenMP, namely AoS and SoA. Surprisingly, the SoA approach outperformed the AoS implementation, providing better results in terms of speedup. The best speedup obtained was 3.2, showcasing the promising capabilities of parallel computing for the Meanshift algorithm and offering a significant performance boost for large datasets that would have been otherwise computationally intensive using only the CPU

6 References

[1] O. Tuzel, F. Porikli and P. Meer, "Kernel methods for weakly supervised mean shift clustering," 2009 IEEE 12th International Conference on Computer Vision, Kyoto, Japan, 2009, pp. 48-55, doi: 10.1109/ICCV.2009.5459204. [Google Scholar]

[2] Robert Strzodka, Chapter 31 - Abstraction for AoS and SoA Layout in C++, Editor(s): Wen-mei W. Hwu, In Applications of GPU Computing Series, GPU Computing Gems Jade Edition, Morgan Kaufmann, 2012

[3] Using OpenMP : portable shared memory parallel programming / Barbara Chapman, Gabriele Jost, Ruud van der Pas.

[4] K. R. Duncan, R. Stewart and G. Michaelson, "Parallel Mean Shift Accuracy and Performance Trade-Offs," 2018 25th IEEE International Conference on Image Processing (ICIP), Athens, Greece, 2018, pp. 2197-2201, doi: 10.1109/ICIP.2018.8451199.