Parallel Computing Project:
Mean Shift Clustering applied to Color Segmentation

Giovanni Burbi
giovanni.burbi@stud.unifi.it

Abstract

The Mean Shift is a non-parametric clustering algorithm that has a quadratic computational complexity, but it has a embarrassingly parallel structure. For this project Java and its parallel mechanisms have been used. We will see the mean shift clustering applied to image processing, more precisely to image segmentation. We will compare the results obtained from the experiments carried out from a sequential and a parallel implementation. Then we will try to optimize the parallel version by using efficiently the cache and avoiding false sharing.

Key words: Java Threads, Mean Shift Clustering, False sharing, Image segmentation, Structure of arrays.

1. Introduction

Non-parametric algorithms can analyze arbitrarily structured feature spaces since these methods do not make assumptions about the clusters. In contrast to the classic K-means clustering approach, there are no embedded assumptions on the shape of the distribution nor the number of modes/clusters. In this project we will use the **mean shift procedure**, proposed in 1975 by Fukunaga and Hostetler. The mean shift is a density estimation-based non-parametric clustering that regards the points in a feature space as a empirical **probability density function** (PDF) where dense regions in the feature space correspond to local maximums, or modes, of the underlying distribution. To estimate the PDF given a set of data points in a d-dimensional space is used the **kernel density estimation** (KDE) method. The idea behind of KDE is, at each iteration of the algorithm, to apply to each point a kernel function that causes the points to shift in the direction of the nearest peak of the KDE surface following the gradient direction. At the end of the algorithm, stationary points correspond to the modes of the distribution (**cluster centroids**) and each cluster is given by the data points in the attraction basin of a mode, where the attraction basin is the region for which all trajectories lead to the same mode.

1.1. Kernel function

For the KDE approach, it is important to choose the **type of kernel function** that will be used which specifies the shape of the distribution placed at each point. There are many different types of these functions, the most frequently used is the **Gaussian kernel**:

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

where x is a point in a space and σ is a smoothing parameter called **bandwidth**.

With this kind of kernel, a KDE requires a single parameter to be tuned, the kernel bandwidth, which controls the **size of the kernel** at each point. Given the bandwidth, the KDE is uniquely determined and so will be its clusters, which can take complex nonconvex shapes. The choice of this parameter is important: a too small value may cause the estimator to show too many details provoking the generation of many small clusters, while a too large value cause to lose much information contained in the sample which, in turn, will result in fewer but larger clusters.

2. Mean Shift Algorithm

Given a set S of N data points x_i , i = 1...N in the d-dimensional space R^d , the shift of each point at each step *t* of the algorithm is computed as a **weighted average** between the considered point and all the others original points (non-shifted points), where the weights are computed according to the given kernel function. The general mean shift algorithm for a given point *x* at a *t* step of the algorithm is as follows:

- Compute the shift vector $m(x^t)$:

$$m(x^t) = \frac{\sum_{i=1}^{N} x_i K(x_i - x^t)}{\sum_{i=1}^{N} K(x_i - x^t)} \tag{2}$$

  where $K(x_i - x^t) = e^{-\frac{||x_i - x^t||^2}{2\sigma^2}}$, we use the **euclidean distance** to measure the different of two points, assuming an euclidean space of the dataset.

- The difference $m(x) - x$ is the mean shift, so set:

$$x^{t+1} = m(x^t) \tag{3}$$

- Iterate previous steps until convergence or a number of iteration has been completed

As we can see the mean shift algorithm is **embarrassingly parallel** so the computation of $m(x^t)$ can be done simultaneously $\forall x \in S$ and $\forall t \in \mathbb{N}$.

3. Image Segmentation

A visual application of mean shift clustering is **image segmentation**. Segmentation is a process that partitions an image into homogeneous regions, this can be accomplished by clustering the pixels in the image. At the end, data points visited by the mean shift algorithm converge to a **mode**, automatically separating clusters of arbitrary shape. The number of significant clusters present in the feature space is automatically determined by the number of modes detected.

The first step is to represent an image as **points in a space**. There are several ways to do this. One way is to map each pixel to a point in a color space using the **RGB pixel values** extracted from the image. To obtain a meaningful segmentation, perceived color differences should correspond to **Euclidean distances** in the color space chosen to represent the pixels. An Euclidean metric, however, is not guaranteed for a color space. The space $L^*u^*v^*$ was especially designed to best approximate **perceptually uniform color spaces**. L^* is the brightness coordinate, while the other two are the chromaticity coordinates. So we will take the RGB values from the pixel of an image and convert them in $L^*u^*v^*$ coordinates. Then we will apply the **mean shift algorithm** to these points, each point (pixel) eventually will shift to one of the modes (KDE surface peaks). In this application the **bandwidth value** can be seen as how close colors need to be in the $L^*u^*v^*$ color space to be considered **related to each other**.

4. Implementation

Our implementation of the mean shift algorithm will use a **gaussian kernel** as the kernel function and the **euclidean distance** to measure the distance between pairs of points. Then we need to choose a **stop condition** for the algorithm, one that will allow us to compare in an objective way the execution time of the different versions of the implementation of the algorithm; we choose to set a **fixed number** of iterations. The last element we need to choose is the value of the **bandwidth**; to have a good estimation of this parameter for our set of points, we can use something like the function "estimate-bandwidth" of the sklearn library or we can do it empirically, trying different values and choosing the one that gives a good enough result.

The **first step** is to extract the RGB values from the pixels of the image and convert them to values in the $L^*u^*v^*$ space. These values will be the **origin points** of our dataset and are stored in a list that will remain **unchanged** throughout the entire algorithm. Then we will copy these points in another list, *shifted points*, where **new positions** are stored after each shifting step. Finally, after a fixed number of iterations the list of shifted points will be converted back to RGB values and a **new image** will be created using those points as pixels.

Algorithm 1 Mean shift algorithm

```
procedure MEANSHIFT($original\_points$)
    $shifted\_points \leftarrow original\_points$
    while $k < MAX\_ITER$ do
        for $point$ in $shifted\_points$ do
            $point \leftarrow$ SHIFTPOINT($point, original\_points$)
        $k \leftarrow k + 1$
```

Algorithm 2 Mean shift subroutine

```
procedure SHIFTPOINT($point, original\_points$)
    $num \leftarrow 0$
    $den \leftarrow 0$
    for $orig\_point$ in $original\_points$ do
        $dist \leftarrow$ COMPUTEDISTANCE($point, orig\_point$)
        $weight \leftarrow$ KERNEL($dist, BANDWIDTH$)
        $num \leftarrow num + orig\_point * weight$
        $den \leftarrow den + weight$
    return $num/den$
```

Algorithm 1 and *Algorithm 2* represent the *pseudo-code* of the mean shift procedure. As it can be seen, this algorithm has a **quadratic computational complexity**, so it is computationally expensive and it does not scale well with the dimension of feature space. The computational complexity of mean shift can be written as $O(Tn^2)$, where T is the **number of iterations** and n is the **number of data** points. The most computationally expensive part of the mean shift procedure is the **identification of the neighbors** of a point in a space (as defined by the kernel and its bandwidth). On the other hand the mean shift algorithm is **embarrassingly parallel**, in fact each single point can be shifted independently from the others.

4.1. Sequential version

The sequential version of the mean shift algorithm has been implemented in **Java** and it's similar to the *pseudo-code* seen in Algorithm 1 and 2. To do image segmentation we need to first extract the RGB values from the pixels of an image, convert them in the $L^*u^*v^*$ space and store them. After we have all original points, we can apply the

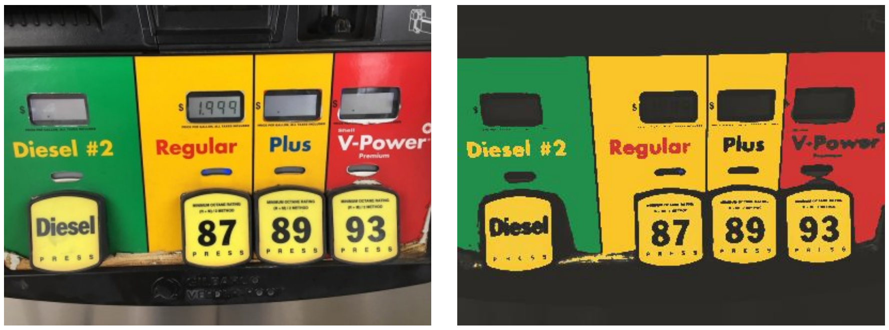

FIGURE 1: *Original image and image obtained applying the mean shift clustering with bandwidth equal to 12 and maximum algorithm iterations equal to 6*

mean shift algorithm, as we seen before. Finally, we can use the shifted points to create a new image that represent the result of the image segmentation with the mean shift clustering, *figure 1*. After extracting the points of an image, there are two ways to store them in memory: Using an **Array of Structures** (AoS) or a **Structure of Arrays** (SoA) layout. The structure in our case is an object that represents a point with three coordinates, its channels. We can have an **object for each of the pixel** and store them in a list (or an array) or we can create an **object that has three list** (or arrays), each representing a coordinate of the space where the the points reside and with length equal to the number of total pixels. *Listing 1* and *listing 2* show how to implement the two layouts.

LISTING 1: *Array of Structures*

```
public class Point {
  private double d1;
  private double d2;
  private double d3;
  ....
}
List<Point> points;
```

LISTING 2: *Structure of Arrays*

```
public class PointsSoA {
  private double[] d1;
  private double[] d2;
  private double[] d3;
  ....
}
PointsSoA points;
```

The Structure of Arrays layout of data is more **friendly to the cache**; it's easily aligned to cache boundaries and it is **vectorizable**, while is easier to call a function on only one element with the Array of Structures even if it can create **cache alignment problems**, *figure 2*.

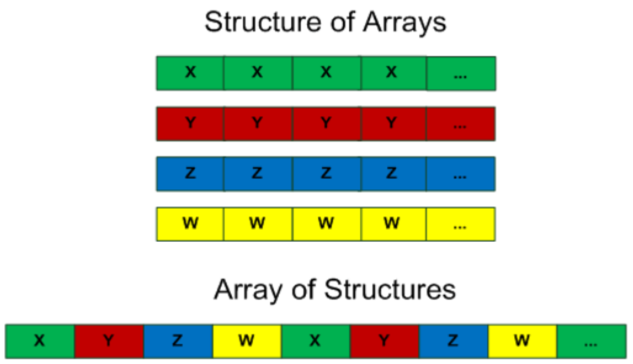

FIGURE 2: *Representation in memory of objects with the array of structures layout and with the structure of arrays layout*

4.2. Parallel version

The parallel version has been implemented with **Java Threads**. As said before, the mean shift algorithm is embarrassingly parallel so we can perform the shift step **simultaneously**. We can also extract the pixels of an image and set the pixels of a new image simultaneously. Therefore, we can identify **three class of threads**: one extracts the RGB values from the pixels of a portion of an image, another applies the mean shift algorithm to a chunk of the points and the last sets the pixels of a portion of a new image to the values of the resulted points obtained from the clustering algorithm. We may set the pixels of a new image directly from the threads that execute the mean shift algorithm but that would violate the **single responsability principle**, so we prefer to divide the tasks in two different threads. This, however, will require an additional step of synchronization before starting the threads that will create the new image.

For this parallel version one important aspect is to **divide the data between threads**. We want the workload to be split among thread as **evenly** as possible. We will use a

static data decomposition because it doesn't require complex coordination to verify if a thread has finished work on his chunk of data. Each thread that will extract data from pixels of the image will receive a block of columns of that image, figure 3. We split the image like this to exploit the **memory burst** when we access the memory. In this way some threads will load in the cache some data that will be used from other threads, improving the performance. The chunks will have the same amount of columns if the total number of columns is a multiplier of the number of threads. Otherwise, we will distribute the remaining j columns, with $j \in [1, N-1]$ and N the number of threads, between the threads with id less or equal to j. In this way we will have some threads responsible for **at most one more column** compared to the other threads. Algorithm 3 shows how the decomposition of data among threads is done. Tid is the id of the thread and nThreads is the number of the threads executed.

Algorithm 3 Data decomposition procedure

```
procedure DATASPLIT($tid, nThreads, matrix$)
    $nElements \leftarrow matrix.width$
    $minChunk \leftarrow nElements/nThreads$
    $surplas \leftarrow nElements - nThreads * minChunk$
    if $tid < surplas$ then
        $chunkSize \leftarrow minChunk + 1$
        $startChunk \leftarrow tid * chunkSize$
    else
        $chunkSize \leftarrow minChunk$
        $offset1 \leftarrow surplas * (chunkSize + 1)$
        $offset2 \leftarrow (tid - surplas) * chunkSize$
        $startChunk \leftarrow offset1 + offset2$
    $endChunk \leftarrow startChunk + chunkSize$
    return $startChunk, endChunk$
```

We need to wait for all the threads that extract data from the image before moving on because, to shift a point, the mean shift algorithm requires to compute the distance between all original points and the point to be shifted. Therefore, only after we have all original points, we can apply the mean shift algorithm, as we have seen before, to chunks of these points simultaneously. We can apply a procedure similar to algorithm 3 to create the chunks of data needed by the threads. After all points have been shifted, we can start the threads that will set the RGB values to the pixels of a portion of a new image assigned to them. To reduce the overhead of creating and destroying threads for each of the task, we created the threads at the start and then we re-used the same threads throughout the program. Only at its end we destroyed the all the threads. To do this, we used the functionality of java's *ExecutorService*

We have two ways to implement these threads: using the **Runnable** interface or the **Callable** interface.

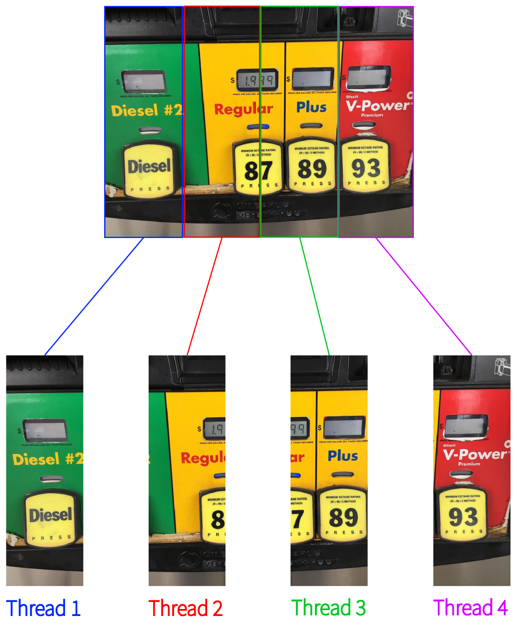

FIGURE 3: *Image decomposition among four threads*

4.3. Runnable implementation

The **runnable interface** doesn't allow the threads to return any values, so the threads must work on a **shared object**. All the tasks we need to perform to the points can be done independently from each other, so we don't incur in race condition when more than one thread tries to access the same data, possibly modifying it. The threads that extract data read the image and store the data obtained in a shared object. So initially, we will need to create a shared object with as many elements as the total number of points that will be extracted, each element initialized with some default value. Each thread will modify the elements inside the chunk assigned to it. We will need to do the same thing with another shared object that will contain the resulted points obtained from the execution of the mean shift algorithm on the original points. To compute the intermediate points, each thread has a local object where it will first copy the chunk of original data that the thread is responsible for and after, it will store the shifted points at each iteration of the algorithm. Finally, we will need a *BufferedImage* shared among threads where each one will set the pixels of the portion assigned to it. To synchronize the three kinds of threads a **phaser** has been used. This sort of implementation can cause a problem when there are writings in a shared object, even if the threads don't directly interfere with each other. This problem is called **false sharing**; it will force some threads to **read again** from memory because their cache has been invalidated by other threads, **reducing the performance**. We will see more about this in the next section.

4.4. Callable implementation

The callable interface, on the other hand, allow us to **return something** at the end of the thread's work. Thanks to that we can create **local object** inside each thread and return them as **futures**. We won't need to have a shared object where to write the results of the algorithm or the extracted points. However, in the main thread, we will need to **merge** the different results in a single object that will be passed to the new set of threads or be used to do some other work. This approach has the advantage of **avoiding the false sharing** because each thread will work only on local variables. An example of the effects of the false sharing, figure 4, shows the **degrading speedup** in the creation of a new image done by increasing the number of threads that work on a shared *BufferedImage* object. We can see that from six threads onwards **the performance decrease even compared to the sequential version**. In fact, the more threads there are, the less columns each thread will work onto, so it happens more frequently that a thread invalidates a cache line of another thread. In that case the thread needs to read again from memory to refresh its cache line, decreasing the performance.

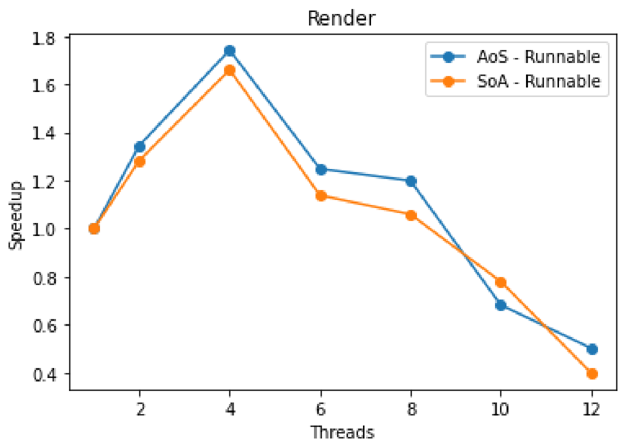

FIGURE 4: *False sharing effect on the creation of an image 4000x2667 done in parallel*

5. Experiments

We have done several experiments to test the different versions of the implementation of the image segmentation with mean shift clustering. For all the experiments **we have set**:

- **Kernel bandwidth** fixed to 12
- The **number of iterations** of the mean shift algorithm to 6
- Experiments time results are the **average over 3 runs** of the application. It includes the extract of data, the clustering algorithm and the creation of the new image with the shifted points.

All tests have been performed on a computer with a Intel© Core™ i7-8750H CPU @ 2.20GHz with 6 cores / 12 threads and java 8.

5.1. Speedup

More than the execution time, the **speedup metric** is important to evaluate the performance of the different implementations. The speedup is defined as the **ratio** between the completion time of the sequential algorithm t_s and the completion time of the parallel algorithm t_p:

$$S = \frac{t_s}{t_p} \tag{4}$$

5.2. Images

In order to perform image segmentation, an image with **strong color variation**, as seen in *figure 1*, has been chosen to produce a good result. The **same image with different sizes** has been used to carry out the experiments, this way we can work with different amounts of points. The sizes of the image used are: 40x30 (1200 pixels), 120x90 (10800 pixels), 200x150 (30000 pixels), 300x225 (67500 pixels), 400x300 (120000 pixels).

6. Results

Now are shown the results obtained from the various tests carried out.

6.1. Sequential versions with AoS and SoA

The results obtained from using different data organizations in memory are shown in *figure 5* and *table 1*.

Number of points	Time AoS(ms)	Time SoA(ms)
1200	418	419
10800	32313	32854
30000	253998	249264
67500	1320531	1319240
120000	4268235	4057801

TABLE 1: *Confrontation between times of the sequential version of the mean shift algorithm with the two different data layouts*

From *figure 5* we can see the trend of the time needed to complete the image segmentation process. With more points with have an **improvement using the Structures of Arrays** layout of data memorization. This improvement will be more evident in the parallel versions.

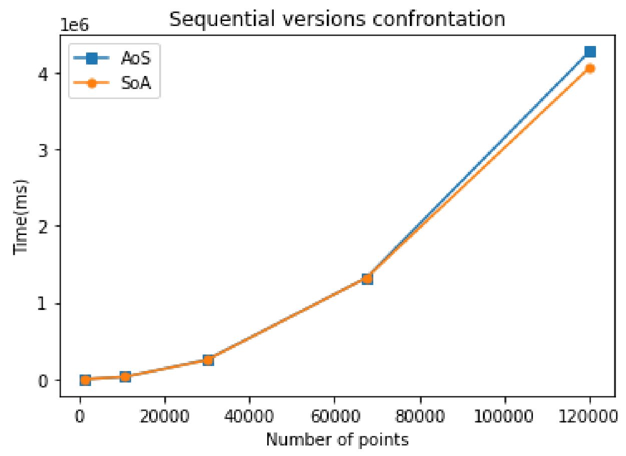

FIGURE 5: *Times confrontation between sequential versions with different data layouts*

6.2. Speedup: Runnable implementation

Some tests were performed to evaluate the performance of the parallel implementation on each image sizes with an **increasing number of threads** (from two to twelve threads). The results with the speedup obtained from the runnable implementation compared to the sequential version for some of the image sizes are shown in *table 2* and *3*.

Version	Threads	Time(ms)	Speedup
Sequential AoS	-	422	-
Parallel AoS	2	233	1.8112
	4	161	2.6211
	6	126	3.3492
	8	101	4.1782
	10	89	4.7416
	12	101	4.1782
Sequential SoA	-	429	-
Parallel SoA	2	223	1.9238
	4	157	2.7325
	6	115	3.7304
	8	98	4.3776
	10	92	4.6630
	12	107	4.0093

TABLE 2: *Results with Runnable implementation for image 40x30 (1200 points)*

These tables show that the **SoA layout has generally a better speedup compared to the AoS layout**. This is because the SoA layout is cache friendlier and it enables better vectorization.

From *figure 6* and *7* we can see the speedup plots for all the tested image sizes as the number of threads increase with the runnable implementation. We can notice that we can obtain a **sub-linear speedup** with this parallel implementation. When there is (relatively) few points we can notice a **decrease of the speedup** because each threads receive **too little work**, the **overhead of scheduling** these threads take more that what we gain from multithreading. Another interesting aspect we can notice is that the speedup for the biggest image increase with the number of threads, but from 10 to 12 threads the speedup is **less steep** than that when using less threads and even when working on a smaller image. This can be the consequence of working with a clustering algorithm with a **quadratic computational complexity**.

Version	Threads	Time(ms)	Speedup
Sequential AoS	-	1320531	-
Parallel AoS	2	671105	1.9677
	4	359766	3.6705
	6	274371	4.8129
	8	218060	6.0558
	10	166082	7.9511
	12	143185	9.2226
Sequential SoA	-	1319240	-
Parallel SoA	2	661254	1.9951
	4	343410	3.8416
	6	250591	5.2645
	8	195101	6.7618
	10	159300	8.2815
	12	138594	9.5187

TABLE 3: *Results with Runnable implementation for image 300x225 (67500 points)*

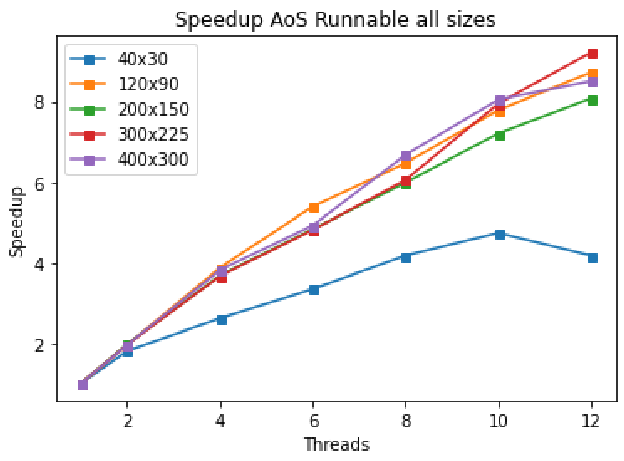

FIGURE 6: *Speedup of the Runnable implementation for all image sizes with AoS layout*

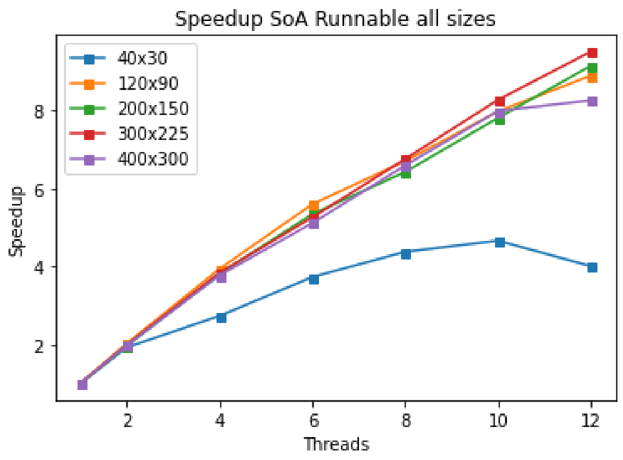

FIGURE 7: *Speedup of the Runnable implementation for all image sizes with SoA layout*

6.3. Speedup: Callable implementation

The results with the speedup obtained from the callable implementation compared to the sequential version for some of the image sizes are shown in *table 4*, *table 5*, in *figure 8* and *figure 9*. The results for the callable implementation confirm the observations done for the runnable implementation.

Version	Threads	Time(ms)	Speedup
Sequential AoS	-	422	-
Parallel AoS	2	224	1.8839
	4	127	3.3228
	6	104	4.0577
	8	86	4.9070
	10	83	5.0843
	12	90	4.6889
Sequential SoA	-	429	-
Parallel SoA	2	222	1.9324
	4	124	3.4597
	6	98	4.3776
	8	85	5.0471
	10	74	5.7973
	12	86	4.9884

TABLE 4: *Results with Callable implementation for image 40x30 (1200 points)*

6.4. Confrontation: Runnable and Callable

We can now compare the two kinds of implementation with the different interfaces. From *figure 10* we can see that the callable implementation tends to have a **better speedup** than the runnable one. This is because it makes more use of

Version	Threads	Time(ms)	Speedup
Sequential AoS	-	1320531	-
Parallel AoS	2	669396	1.9727
	4	349944	3.7735
	6	273601	4.8265
	8	214544	6.1551
	10	159879	8.2596
	12	140012	9.4316
Sequential SoA	-	1319240	-
Parallel SoA	2	667859	1.9753
	4	338245	3.9002
	6	235983	5.5904
	8	190182	6.9367
	10	156706	8.4186
	12	137282	9.6097

TABLE 5: *Results with Callable implementation for image 300x225 (67500 points)*

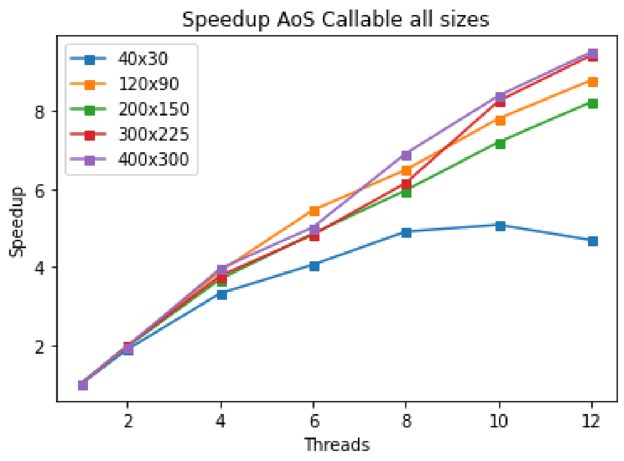

FIGURE 8: *Speedup of the Callable implementation for all image sizes with AoS layout*

local variables and, as anticipated, because of the false sharing that influence the runnable implementation since writing on a shared object by a thread may cause the invalidation of a line of cache from another thread which therefore must read the new value from memory, decreasing the performance.

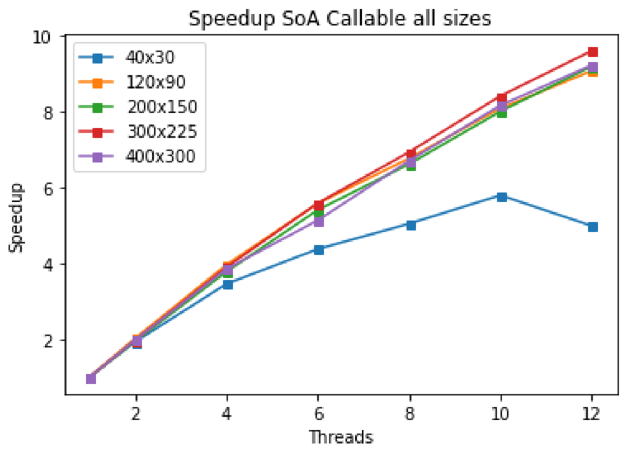

FIGURE 9: *Speedup of the Callable implementation for all image sizes with SoA layout*

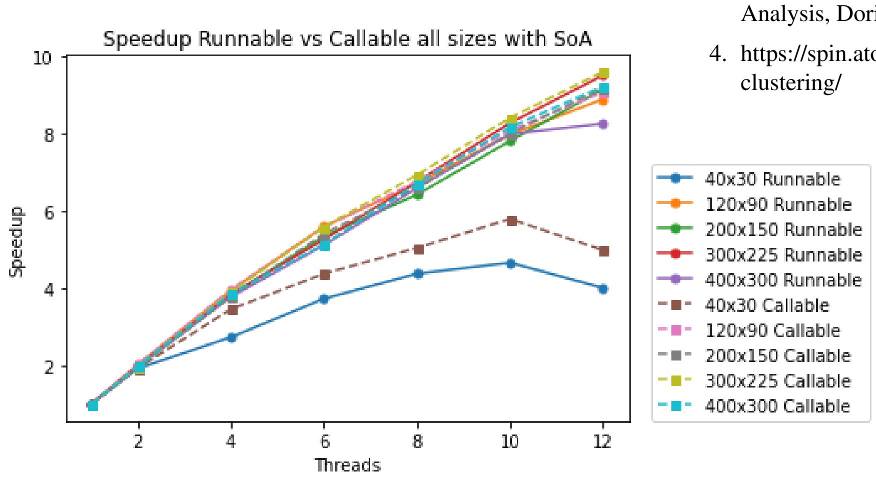

FIGURE 10: *Confrontation between Runnable and Callable implementation for all image sizes with SoA layout*

7. Conclusion

We have seen the mean shift clustering applied to **image segmentation**. This clustering algorithm has a **quadratic computational complexity** but it is also **embarrassingly parallel** so we can use the multithreading of Java to improve its performance. We have seen that it is worth implementing a parallel version of the mean shift clustering when we need to work with a sufficient amount of points. This way we will obtain a good speedup even with many threads. We have also seen that using a **Structure of Arrays** layout of data in memory can give better performance with respect to the standard Array of Structures layout. Finally, we have seen that an implementation with the **callable interface** gives better results because the threads can work only with local variables. Performing the image segmentation with the mean shift clustering using the parallel capabilities of Java Threads, implementing the Callable interface to define the threads and using the Structure of Arrays data layout, can result in a speedup **up to 9.6x** respect the sequential implementation.

8. Resources

The code is available on **GitHub**:
`https://github.com/GiovanniBurbi/MeanShift.git`

9. Bibliography

1. A review of mean-shift algorithms for clustering, Miguel A. Carreira-Perpinàn
2. Mean Shift, Mode Seeking, and Clustering, Yizong Cheng
3. Mean Shift: A Robust Approach Toward Feature Space Analysis, Dorin Comaniciu, Peter Meer
4. https://spin.atomicobject.com/2015/05/26/mean-shift-clustering/