
UNIVERSITÀ
DI PAVIA

# 2D K-Means clustering parallelization

Advanced Computer Architecture lab project

Luca Todaro
Rasha Zieni

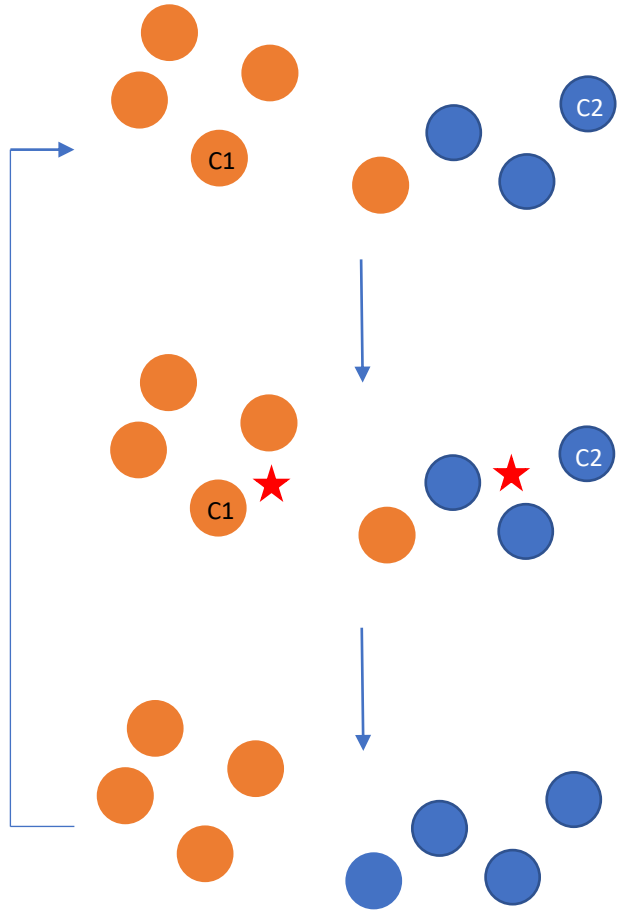
# Outline

- Introduction
  - Algorithm overview
- Serial implementation
  - Design choices
  - Validation
- Parallel implementation
- Data gathering and analysis
  - Test datasets
  - Local experiments
  - Google Cloud Platform
- Conclusion

# Algorithm overview

The goal of K-means is to divide a given set of points into K different groups that share the same characteristics by assigning them to different clusters.

1. Pick K random points from the dataset
2. For each point, compute the distance between the point itself and every centroid and assign it to the nearest
3. Recompute centroids on newly formed clusters and repeat from step 2



# Algorithm overview

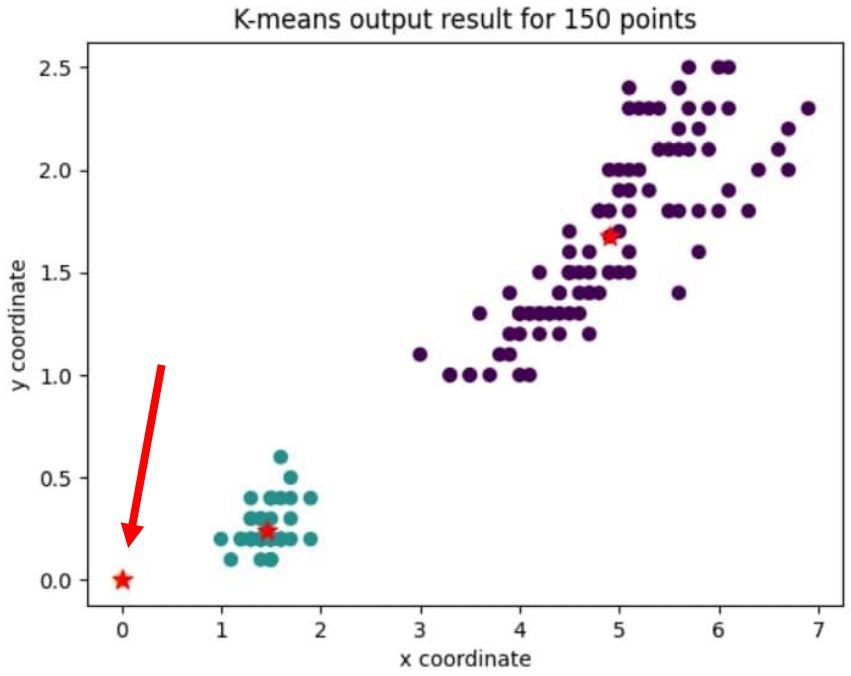
The algorithm converges once it satisfies one or more of the following conditions:

- A specified number of iterations is reached
- At least one centroid does not change its position or migrate less than a threshold
- Points do not change their cluster assignation
- Sum of distances is minimized (inertia)

Random choice of centroids: this may cause the algorithm…

1. To converge in a different number of iterations
2. To end up with different cluster assignations

Centroids must be picked among the data points!!!



# Serial implementation - design

To keep the code as bare minimum as possible but without sacrificing readability and respecting healthy coding principles, we took advantage of C++ classes. Two classes are defined:

- **Point:**

  **A Point object models a point in a 2D space**

  - x_coord, y_coord: the x and y coordinates of a point in the Euclidean plane
  - cluster_number: an identifier to assign a point to its corresponding cluster
  - Setter and getter methods, as all the class members are private

- **Cluster:**

  **A Cluster object represents a collection of points**

  - pivot: a cluster has a pivot which is a Point. This one stores the actual coordinates
  - new_pivot: a buffer Point object as the pivot must be updated only at the end of an iteration
  - dimension: how many points belong to the cluster. Needed because of computing its center of mass
  - Functions to add a point to the cluster, to update it and to clear it from all the points such that it gets ready for a new iteration

# Serial implementation - design

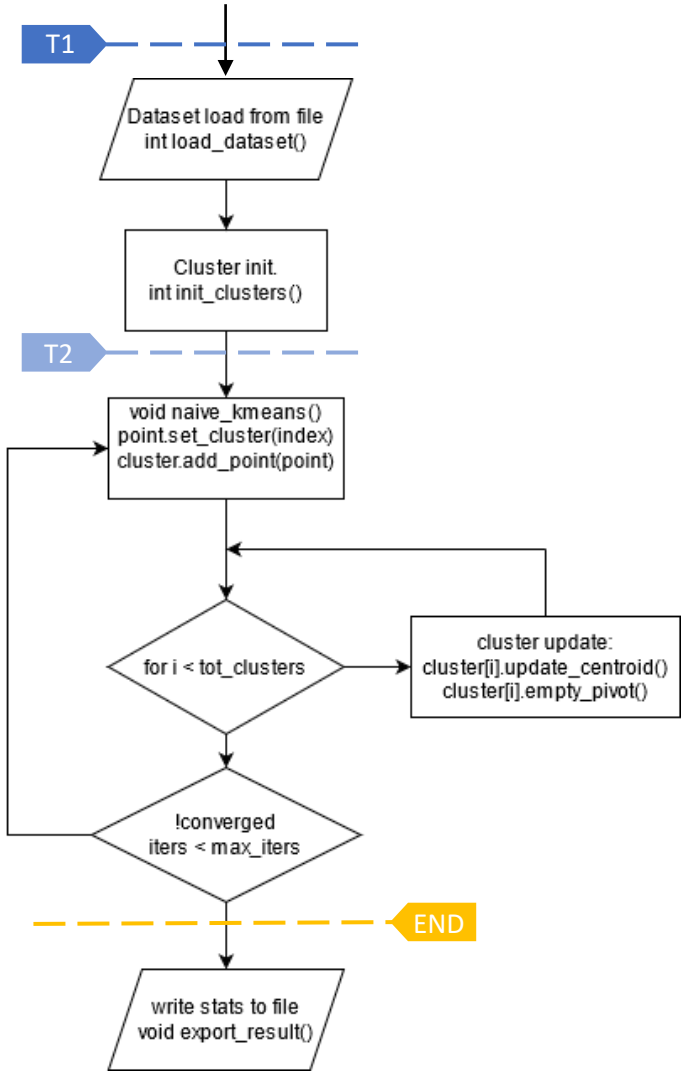
Data types involved in our script are very simple as well; because we usually worked inside an IDE, we didn’t feel the need of command line parsing and by sacrificing it we hugely increased code simplicity

- **Array of points and clusters:**
  - Faster than Vector construct
  - Static allocation because of the nature of the algorithm
- **Three given data**:
  - Number of clusters
  - Number of points
  - A maximum number of iterations

In our case it is also possible to specify the dataset type to use and, in the parallel implementation, the number of threads.

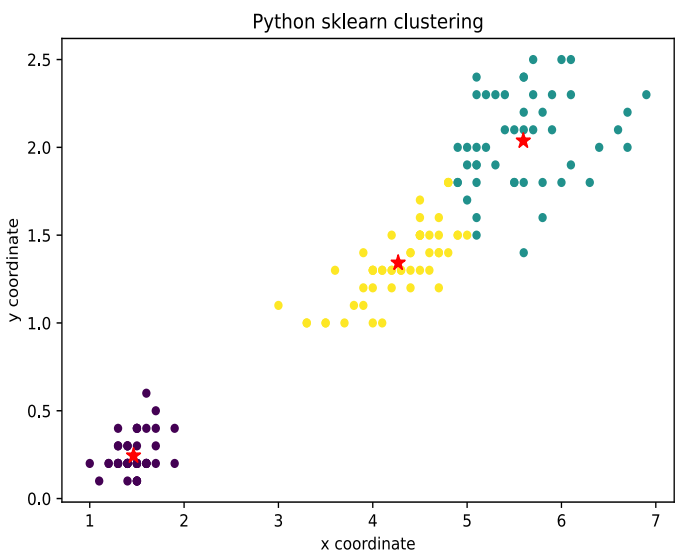
# Serial implementation - design

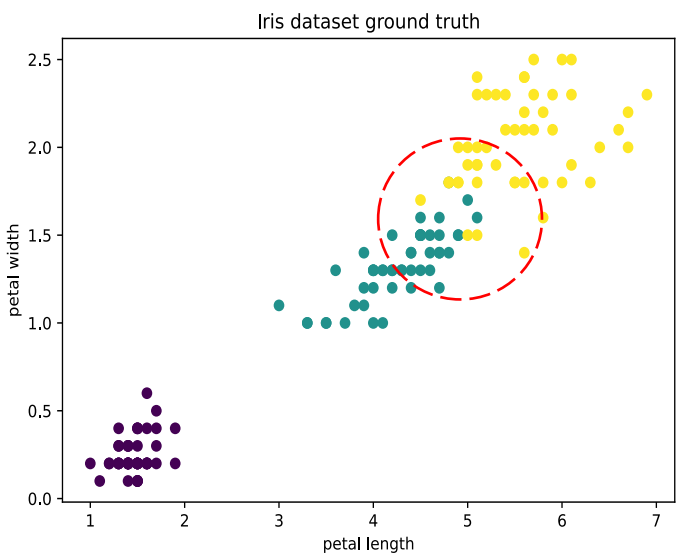
1. Dataset loading from file
2. Clusters initialization
3. In the while loop:
   - K-means computation
   - Update all centroids by setting new coordinates as current
   - Clear the pivots to prepare the clusters for a new iteration
4. Dump timings to file and export the clustered dataset writing them to another file

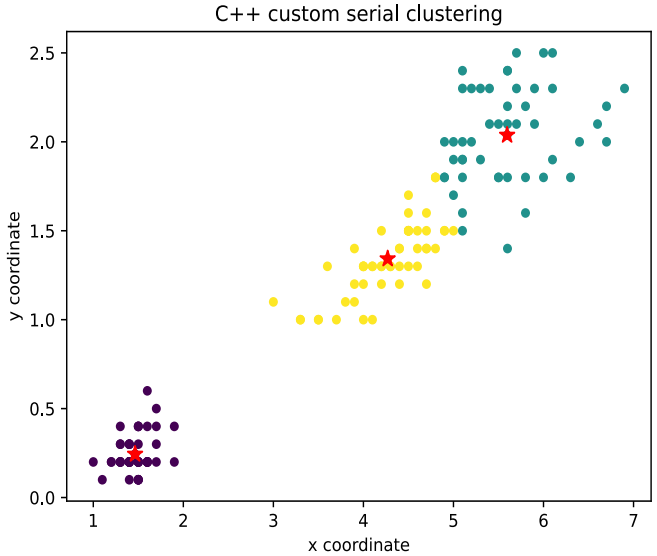


# Serial implementation - validation

To validate the accuracy of our serial implementation we tested against the K-Means Python implementation built in Scikit library using the Iris dataset and some synthetic datasets featuring different numbers of points drawn from gaussian distributions

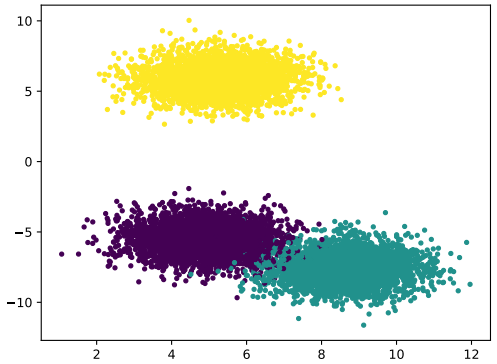


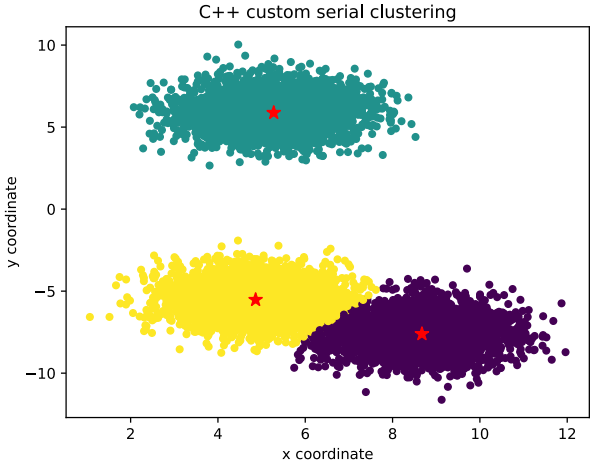



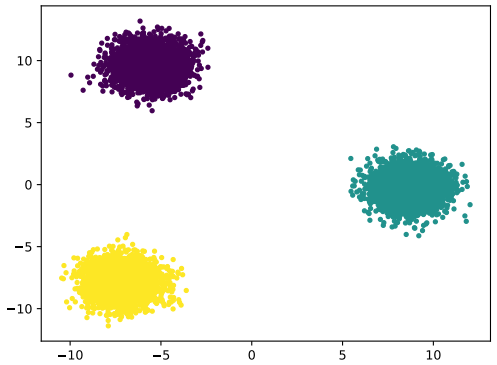
# Serial implementation - validation

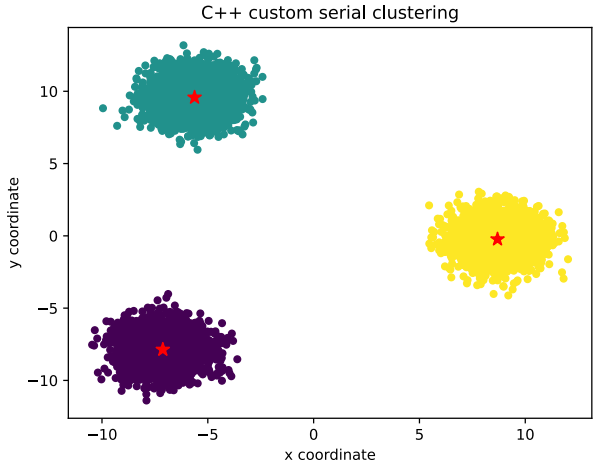
These results are instead obtained from the synthetic dataset we built from scratch, but the number of optimal clusters is known




Still struggles to classify on overlapping datasets, but if they are completely disjointed, the result is flawless



# Parallel implementation

We first had to identify relevant code areas to parallelize; to do so, we compared timings in most impactful code areas. Clusters are fixed, because we had to consider the computational load a higher number of those would introduce.

| Points, clusters | Serial code segments | | |
|---|---|---|---|
| | *k-means loop iteration* | *Cluster update* | *Distance computation* |
| **10000, 10** | 17.184 ms | 0.009 ms | 10.749 ms |
| **50000, 10** | 76.408 ms | 0.011 ms | 47.148 ms |
| **100000, 10** | 148.816 ms | 0.013 ms | 94.575 ms |

Out of a single kmeans iteration, 60% of time is spent to compute Euclidean distances and the rest to assign a point to a cluster by recomputing x and y coordinates on the buffer pivot object

# Parallel implementation

We parallelized the code using OpenMp library

- Parallel outer loop
- Static scheduling, as each thread has the same amount of computations
- Private variables on each thread: min_distance and min_cluster_index
- Shared variables are the array of points in memory and the array of clusters
- OpenMp critical directive to prevent racing between threads when updating a cluster

```
void naive_kmeans(){
   double min_distance;
   int min_cluster_index;

#pragma omp parallel private(min_distance, min_cluster_index) num_threads(threads)
   {
#pragma omp for schedule(static)
      for(int i=0; i<num_points; i++){
         min_distance = distance(points[i], clusters[0]);
         min_cluster_index = 0;
         for(int j=0; j<num_clusters; j++){
            double tmp_distance = distance(points[i], clusters[j]);
            if(tmp_distance<min_distance){
               min_distance = tmp_distance;
               min_cluster_index = j;
            }
         }
         points[i].set_cluster(min_cluster_index);
#pragma omp critical
         {
            clusters[min_cluster_index].add_point(points[i]);
         }
      }
   }
}
```

# Parallel implementation

Tradeoff between a single critical region and three atomic directives: timings do not differ between the two solutions, but…

```
#pragma omp critical
        {
           clusters[min_cluster_index].add_point(points[i]);
        }
```

We cannot use atomic in our case, as the operation on which we would like to apply it are not unary!

```
  void add_point(Point point){
#pragma omp atomic
    this->new_pivot.set_x(new_pivot.get_x() + point.get_x());
#pragma omp atomic
    this->new_pivot.set_y(new_pivot.get_y() + point.get_y());
#pragma omp atomic
    dimension++;
  }
```

**…NOPE!**

The only way to implement atomic directive would be not to use two object Point inside the class but instead, to explicit the coordinates

```
class Cluster{
public:
  Cluster(){}
  …
private:
  Point pivot;
  Point new_pivot;
  int dimension;
};
```

```
class Cluster{
public:
  Cluster(){}
  …
private:
  double x, y;
  double new_x, new_y;
  int dimension;
};
```

# Data gathering and analysis

Data was gathered by logging the run output to a csv file including the execution (s/p), language used (p/c), dataset type, the number of points, clusters, threads, iterations, wall and k-means time

```
1622320725,s,p,blobs,50000,2,1,12,0.34308,0.10472
1622320744,p,c,blobs,30000,2,6,123,0.28324,0.0748
1622320751,p,c,blobs,30000,2,2,3,0.27526,0.07679
...
```

Two assumptions on the results were made before analysis:

1. Timings are dependent both on the number of clusters and on the number of data points.
2. An increase in data points should cause a bigger impact.

# Data gathering and analysis

We first made some assumptions about the maximum theoretical value for the expected speedup using Amdahl's law:

$$S_{speedup} = \frac{1}{(1 - F) + \frac{F}{N}}$$

Where F is the fraction of parallelizable code and N is the number of processors.

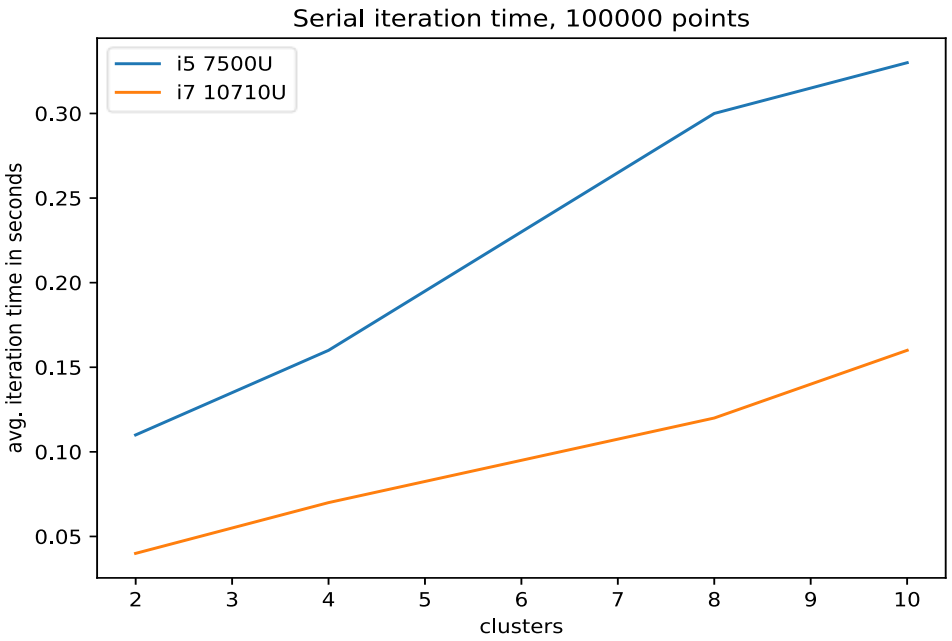
- F scales up with problem size so the exact value of it cannot be defined.
- Maximum speedup expected regardless of the number of cores is supposed to be *at least* a factor of 10 at F = 0.9 (worst case)

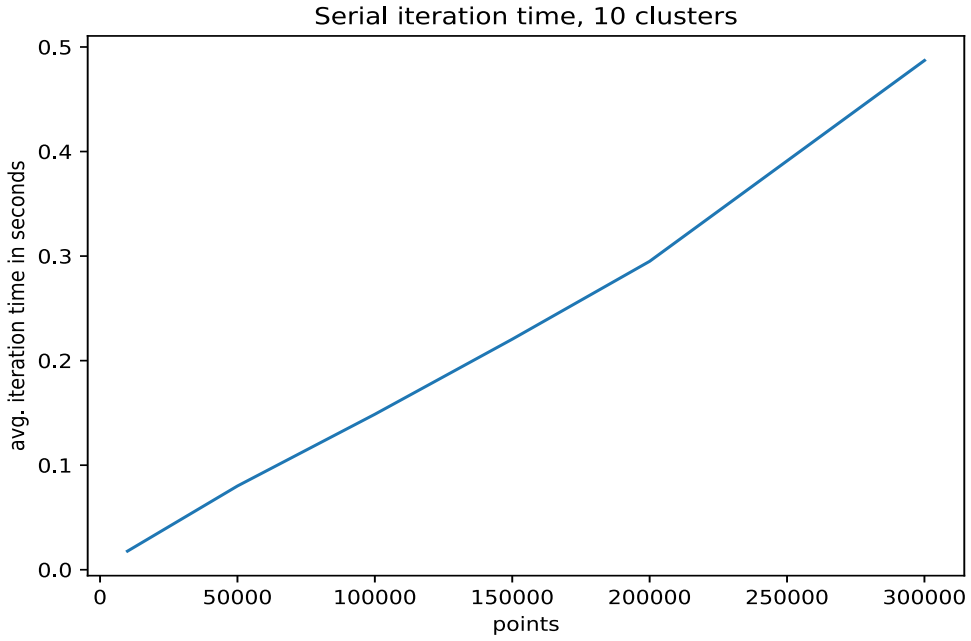
| Points, clusters | Number of processors (N) | F factor | Theoretical speedup (S) |
|---|---|---|---|
| **100000, 10** | 2 | 0.95 | 1.9 |
| | 4 | | 3.5 |
| | 12 | | 8 |
| **300000, 10** | 2 | 0.97 | 1.9 |
| | 4 | | 3.7 |
| | 12 | | 9.6 |

# Local Experiments

**Benchmarking Serial implementation:**

- Testing the difference in time between two processors from Intel.
- Comparison between average k-means iteration time with changing number of points or clusters.
- **Notice**: varying the number of data points is more impactful than increasing the number of clusters.

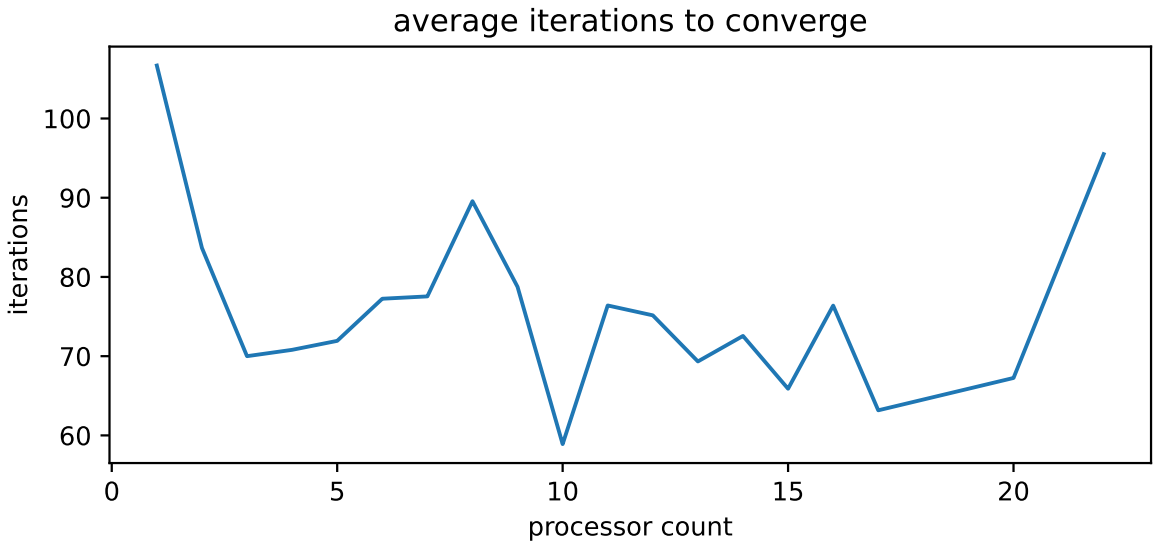


# Local Experiments

**Benchmarking parallel implementation:**

- No correlation between number of processors and number of iterations.
- Experiments were run multiple times for each configuration.
- Mean values computed between all the gathered data.



# Local Experiments

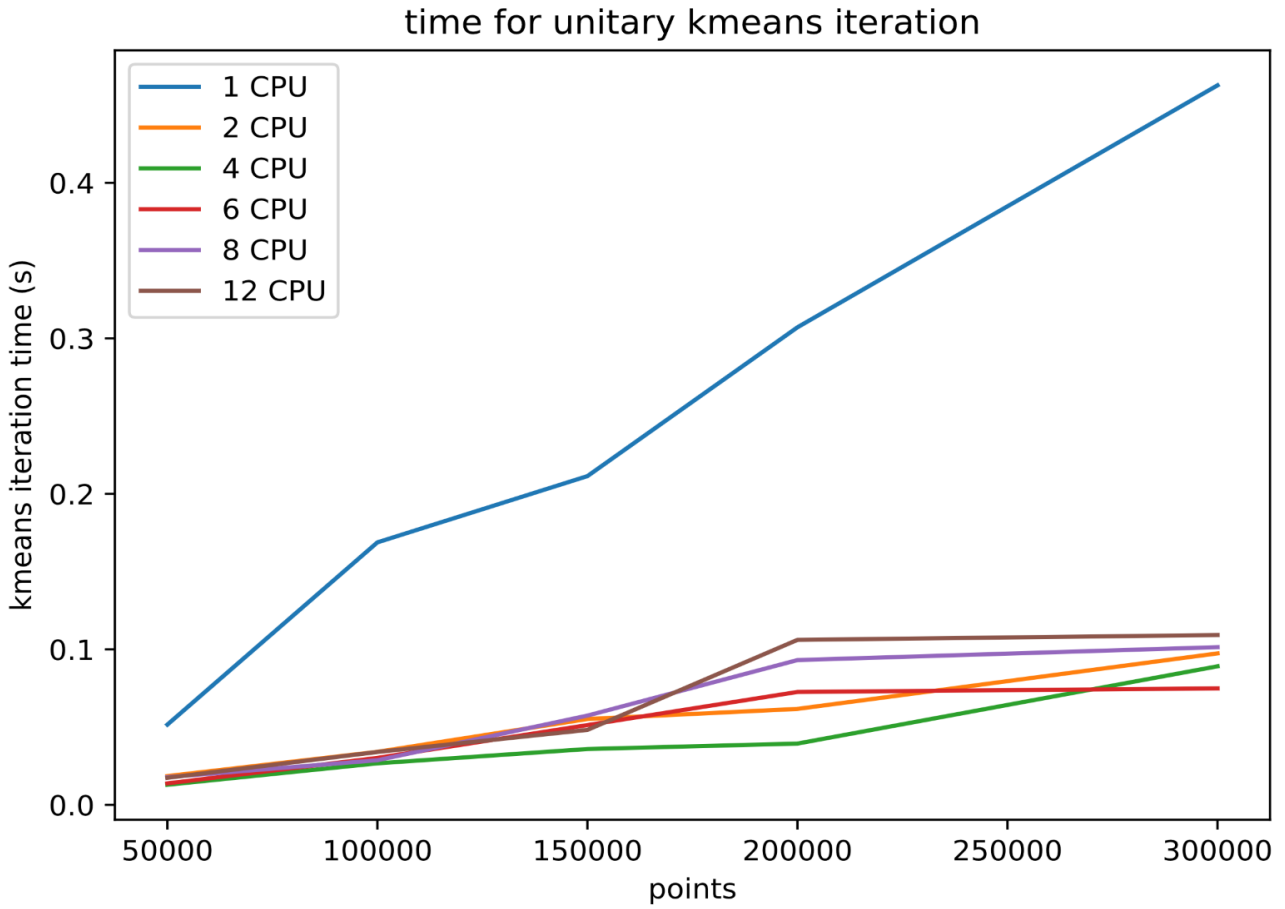
**Benchmarking parallel implementation:**

**The experiment:**

Changing the number of points for each processor count to test how parallelism handles the increase of data to process.

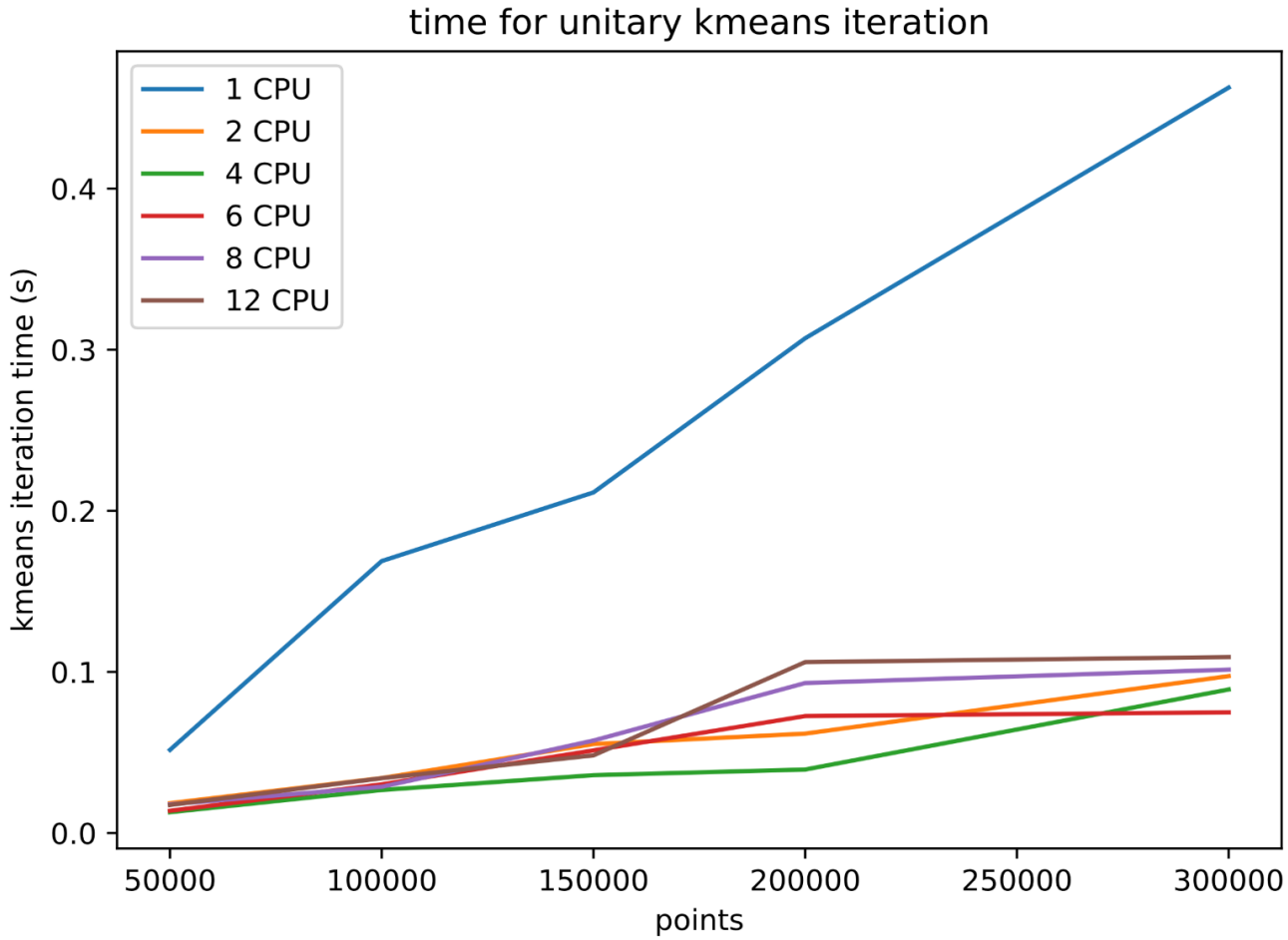
**Results:**

- Almost perfect linear increase in serial mode.
- Good results for runs with 2 and 4 cores up until 200000 data points.



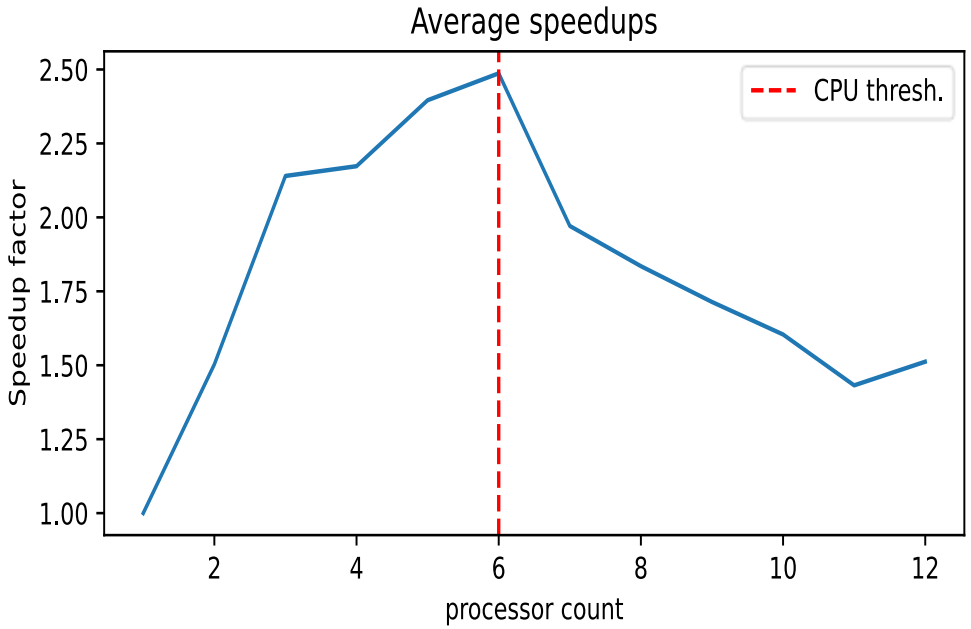
# Local Experiments

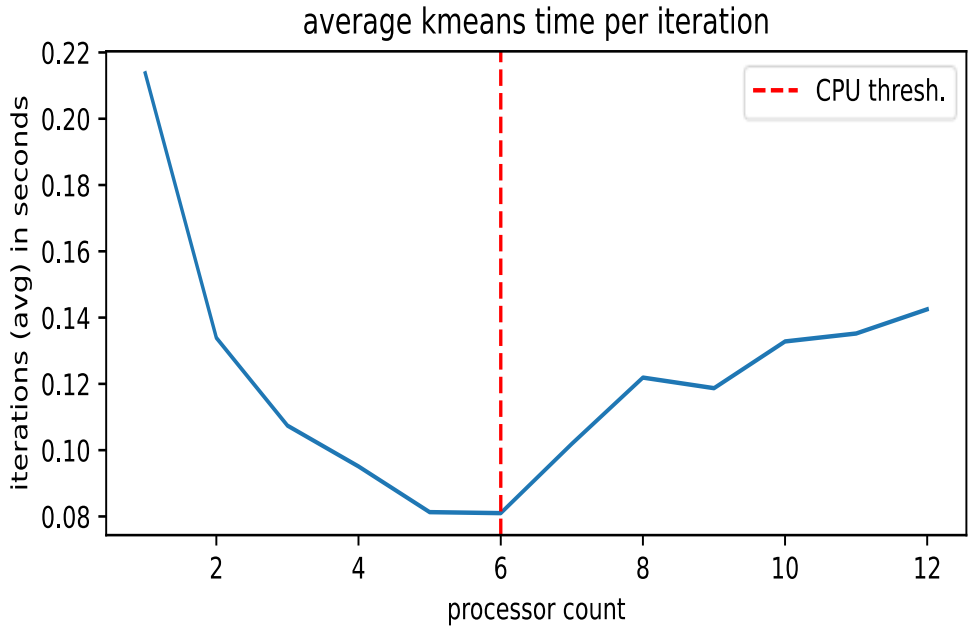
- Struggles with 2 and 4 cores after 200000 points into handling the load increase After that threshold.
- In the region between 150000 and 200000 points they almost stay constant.
- Parallel runs with 6 cores scale better when data points are more than 200000.
- The performance gain starts decreasing after 6 cores.
- After 6 cores with hyperthreading no significant improvement.



# Local Experiments

- High performance gain with parallelism of almost 2.5 times the serial implementation took to complete.
- Decrease in the performance after 6 cores.
- The critical section is not negligible with 12 threads.

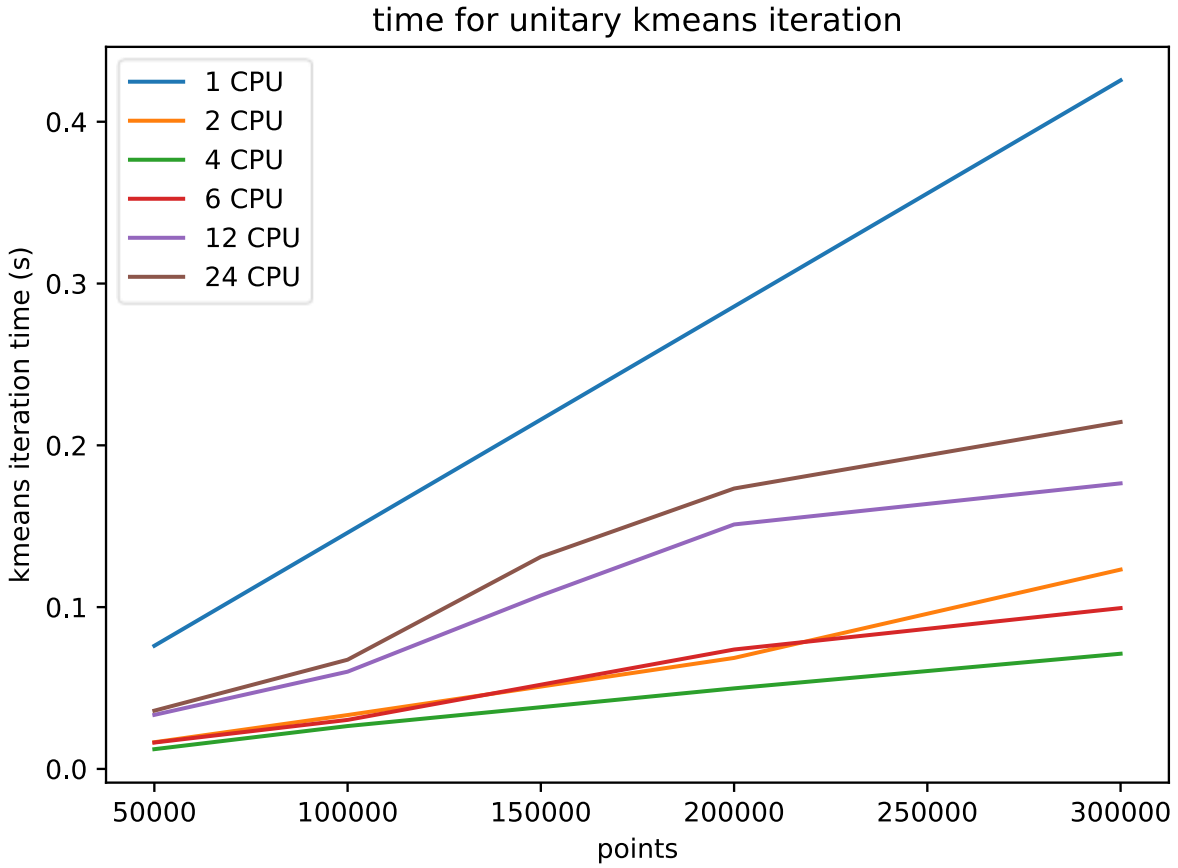


# Google Cloud Platform Experiments

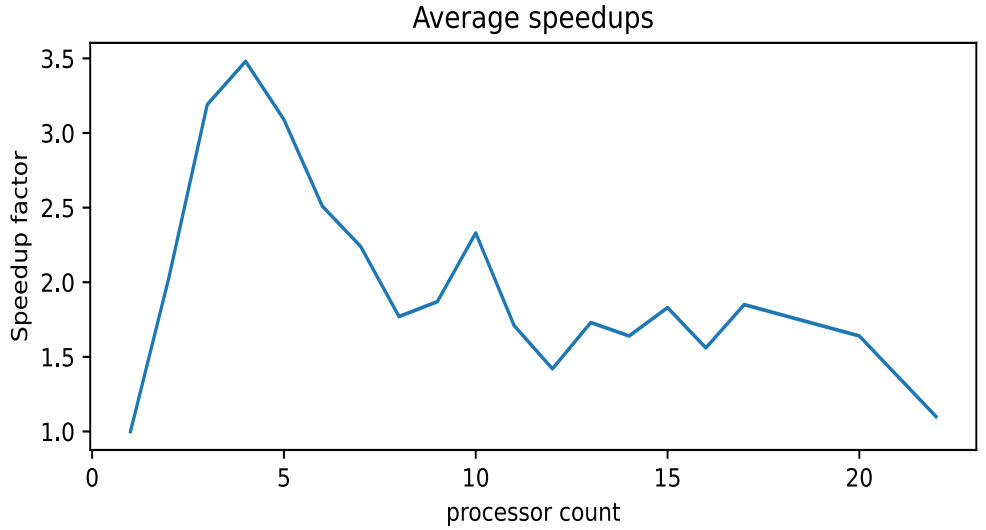
We ran the experiments on a virtual machine with 24 vCPU:

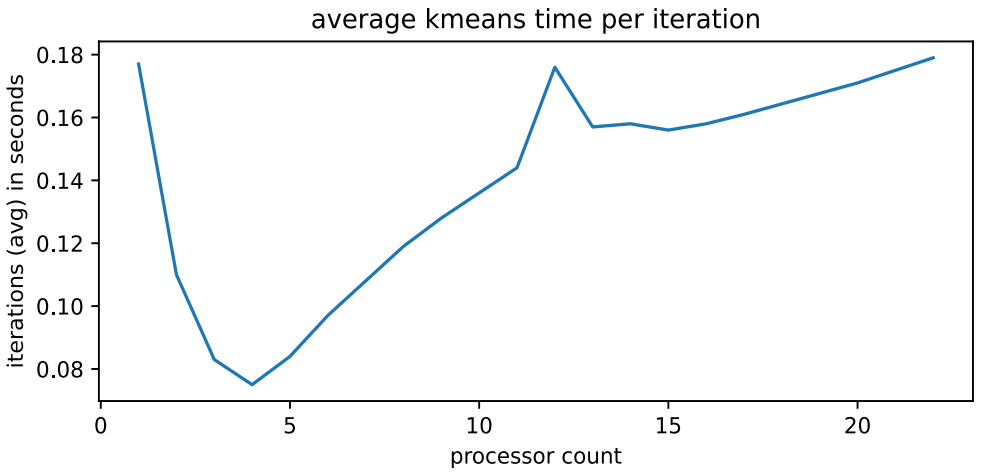
- Gradually increasing the number of processors inside the code.
- Scaling up until 6 CPUs the same results obtained as the local experiment.
- Serial runs take less time than local experiment on 300000 points dataset and almost the same time on the one with 50000 entries.



# Google Cloud Platform Experiments

- A big increase up until 4 processors then the line decreases slowly after this threshold.
- Number of cores that provided the best performance are 4 against 6 processors with our computer.
- The best timing we achieved was the same (0.08 s) as local although the serial time to complete a single iteration is a bit higher.



# Conclusion

- K-means is nondeterministic so It is important to start from a good serial implementation as it will massively affect the results obtained from parallelizing it.
- Amdahl's law and theoretical hypotheses on the maximum speedup expected were confirmed in our local and remote experiments.
- Hypothesis that parallel performance is also a function of problem size  was confirmed.