Report – k-means clustering project

Andrej Perković

May 2022

Abstract

In this report, I document all the relevant information regarding the logic and the implementation of my project. The challenge at hand was to use the k-means clustering algorithm to determine sub-optimal locations for garbage treatment facilities in Germany.

Introduction

For this project, we are given a data set of garbage collection facilities in Germany with their geo-location and capacities. There are 11 000 data point sets provided by the teachers. Since our the data points also include the respective capacities, we are actually dealing with a weighted k-means clustering problem. That is, we do not just take the distance into consideration when decide on the placement of the treatment facilities. This report includes the following sections:

Methodology
: Explanation of the mathematics used for the calculations

Implementation
: Details about the algorithm was translated to a Java program, that is the sequential, parallel and the distributed implementation

Testing
: Comprehensive benchmarks and comparison between different implementation of calculations

Conclusion
: Final thoughts regarding different modes of computation and limits thereof

1 Methodology

The k-means clustering is a well known and highly utilized algorithm, especially in data science. There are even many variations regarding constrains and exact formulae used. What they all have in common is that they are comprised of two steps - *assignment step* and *update step*.

In the assignment step, we assign to each data point the appropriate centroid, that is the closest one. The Euclidean distance formula for calculating the distance is used. We assign point p_i to cluster C_j if it holds that:

$$j = \arg\min_l \{\sqrt{(p_i(x) - C_l(x))^2 + (p_i(y) - C_l(y))^2}\}$$

Now that each point belongs to one of the clusters, we proceed to the update step. We update the location of the centroid of the given cluster based on the collection of the points assigned to it. Instead of just simply finding the mean of the x and y coordinates of the points as in the "naive" version of the algorithm, I used the version that also puts weight on the capacities of the points when calculating, in the following way:

$$C_j(x) = \frac{\sum\limits_{x_i \in C_j} x_i \cdot w_i}{\sum\limits_{x_j \in C_j} w_j}$$

The new y coordinate of the centroid is calculated similarly. This way the calculation is biased towards the "heavier" points, which reflects practical needs of placing the treatment facilities closer to places with more significant accumulation of waste.

Since it's proven that k-means algorithm does not necessarily terminate, we will have to make a hard-coded limit on the number of cycles to guarantee termination. The stopping condition is a design choice. We should either stop when no datapoint is assigned a different cluster than in the previous cycle or when the centroids don't change their coordinates compared to those in the previous iteration, or both. I decided for the one determined by the changes in cluster coordinates.

Finally, we need to consider ways of initialization. I chose to do the Frogy method of randomly choosing k observations from the dataset and using these as the initial means. It's the preferred method for the simpler versions, like the one I am dealing with in this report.

2 Implementation

2.1 Graphical interface

The Java application is implemented with the `Mavean` environment. I used the `JXMapViewer2` library for the maps and the `Spring` framework for the GUI. The latter includes a map initially centered in Koper that reframes the view of the map to include all the datapoints. User is able to choose between sequential and parallel mode of computation and is informed about the time in milliseconds and the number of cycles needed. Due to challenges of running a distributed program in `Java`, the distributed mode of computation is not supported in the combination with the GUI. It is meant to be run from the command line, but it displays the results in on a GUI with a map. Clusters are colored differently and centroids' sizes are scaled relative to each other.

2.2 Calculations

There is a class for each of the modes - `Computation`, `ParallelComputation` and `DistributedComputation`, respectively. Other helper classes are shared between `Computation` and `ParallelComputation`. They, for example, facilitate the import of the *.csv* files (`SiteLoader.`), adapt datapoints to be displayed by the `JXMapViewer2,` (`SwingOverlayWaypointPainter`), initialise the GUI (`Window`), etc. `DistributedComputation` is more self-contained. More on this below.

2.3 Run mode

Both parallel and distributed computation adapt to the environment they are ran in. The former with the help of the `Runtime` class, allowing us to get the machine's number of processes dynamically and adapt the execution accordingly. The latter by including the `"$(sysctl -n hw.ncpu)"` as a parameter for the number of processed for the `MPI` executable, also allowing for dynamic calculation of processor on macOS machines. Moreover, this command checks for the number of available threads which is more than the number of processors in case they support multythreading. All modes use updates on centroids as the stopping condition.

2.3.1 Sequential mode

The sequential mode is straight forward. I follows the basic idea of the algorithm. There are flags checking on the changes in the update of centroid. At the end of each cycle, the status of update of each centroid in the form a `boolean` is iterated. In case there is a flag set to `true`, flag singaling continuation - `changed` - is set to `true`. This way we assure the looping stops only when all flags are `false`.

2.3.2 Parallel mode

This mode was implemented using the thread pool principle with the help of `Executors` class. This allows for dynamic control of threads, so there is less possibility for human error. Synchonization was done with the `CountDownLatch` instance called `barrier`, reinitialized before each assignment and update step.

For the binding step, each thread gets an approximately equally sized chunk of `Site` collection to deal with. To do this, all threads need to know location of current centroids, but they can safely share this since they only read the values in this step. No critical sections here regarding sites, since threads are accessing a partition of the `Site` list, i.e. disjoint sets. On the other hand, the parallel program can get into the race condition in the part of the code where a `Site` object is appended to a cluster's list of the given objects. For this reason, I performed the insertion of sites to appropriate lists sequentially after the computations are done.

For the update step, we call the method `updateCentroid()`, which creates a `Runnable` for each cluster and sends it to the executor. We have a helper `boolean` array for storing the information whether the cluster i was updated at location i. Here, we also don't have a critical section, since no two threads access the same position in the array.

2.3.3 Distributed mode

This mode is more "independent", or better put, "self-contained". Due to the nature of message passing and `MPJ`'s underlying implementation in `C`, is not possible to pass complex structures between processes, like `CLuster` and `Site`. We can only easily send the standard types. For this reason, I decided to "serialize" the `Cluster` and `Site` arrays. I transform them into `double` arrays, `clusterBuffer` and `siteBuffer` respectively. For `Cluster` transformation, I extract the *id*, *latitude* and *longitude* of each instance. Hence, for cluster i, we have its *id* at position i, its *latitude* at position $i+1$ and its *longitude* at position $i+2$ in `clusterBuffer`. Similarly for the sites, I store the *id* at position i, *latitude* at $i+1$, *longitude* at $i+2$, *weight* at $i+3$ and $clusterID$ at $i+4$ in the `siteBuffer` for the site i. Processes loop as long as the stopping condition is not satisfied. To check this, there is a separate buffer for flags, since each process can get a varying number of clusters for centroid updating. Process 0 acts as a coordinator and completes the setup, that is the transformation into `double` arrays. After that, we start looping.

Since the first step in the algorithm is assignment, the coordinator first broadcasts `clusterBuffer` and then scatters the `siteBuffer`. I used `Scatterv` for the latter, since the number of sites need not be divisible with the number of processes running.

For the update step, I decided to implement it in the way that all the processes loop through the entire `siteBuffer`, but only perform calculations on instances whose $clusterID$ corresponds to the cluster instance the given process is assigned. For this reason, I used the `Allgatherv` on the `siteBuffer`

right after the assignment step, but then I used `Scatterv` on `clusterBuffer` to assign approximately equal number of clusters to each process. In addition, I also scatter the flag buffer that store the information whether the cluster was updated.

Finally, the coordinator checks whether there is a `true` flag and broadcast the information. The loop is repeated as long as all the flags are not set to `false` or the limit of 1000 loops is reached.

2.4 Data points

The main source of data points is the *.csv* available on Moodle. It contains 11092 point with localities' coordinates and respective capacities. In case a user asks for a bigger data set, random points are generated with the constraint that longitude, latitude and the capacity is within the range of the data point from the aforementioned file.

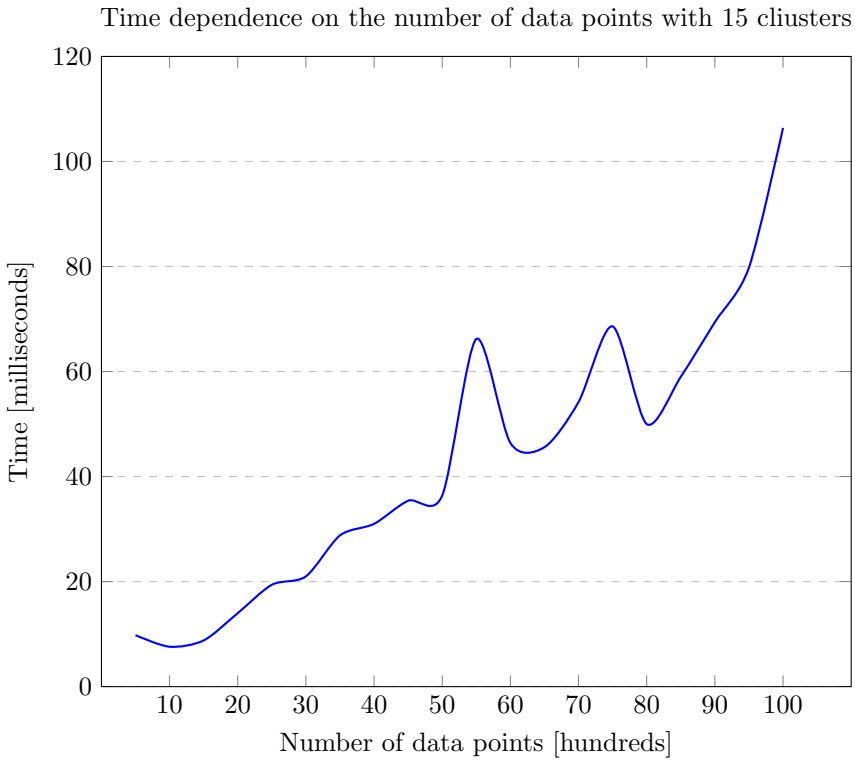
3 Testing

Now, we will turn to the testing result of each of the modes. Time points only count the time necessary for the calculations, hence they exclude the graphics and data setup.

3.1 Varying number of sites

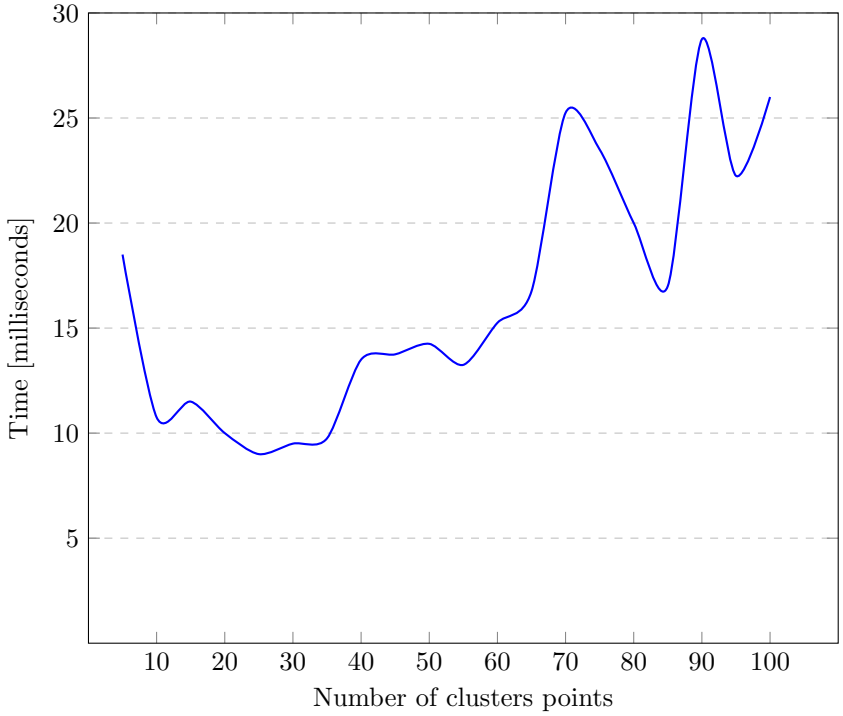
The following results are given for the constant number of clusters set to 15. Data points are the average of 5 samples.

3.1.1 Sequential mode

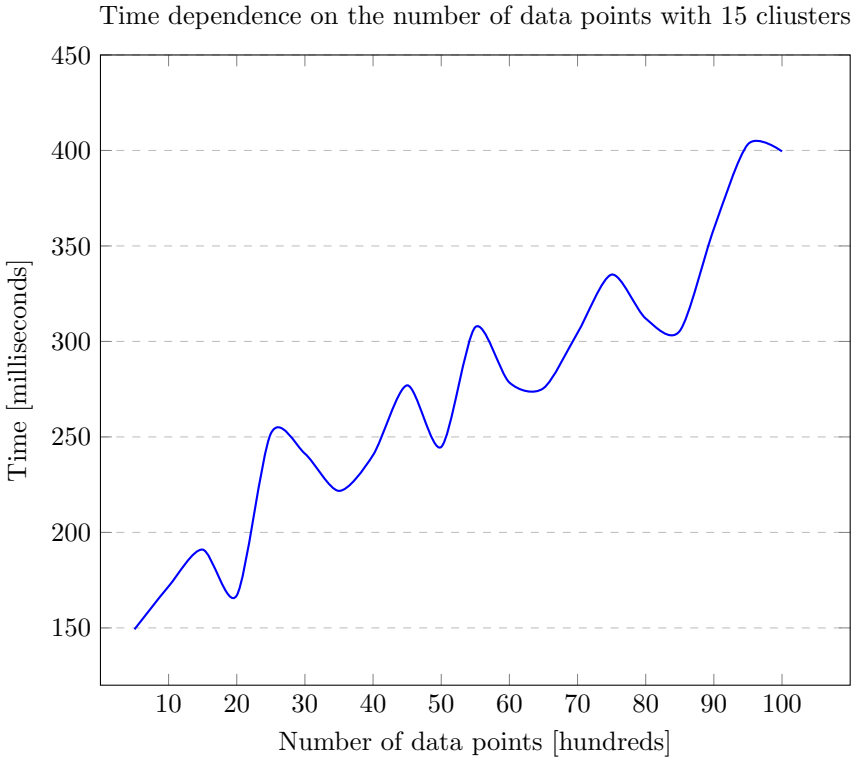


3.1.2 Parallel mode

Time dependence on the number of clusters with 30 000 data points



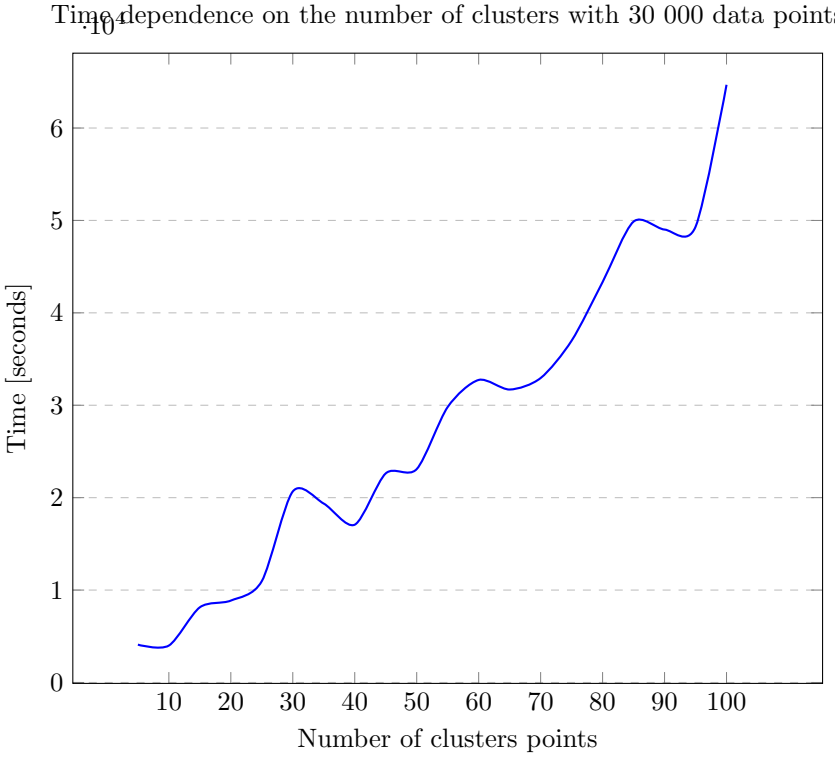
3.1.3 Distributed mode



3.2 Varying number of clusters

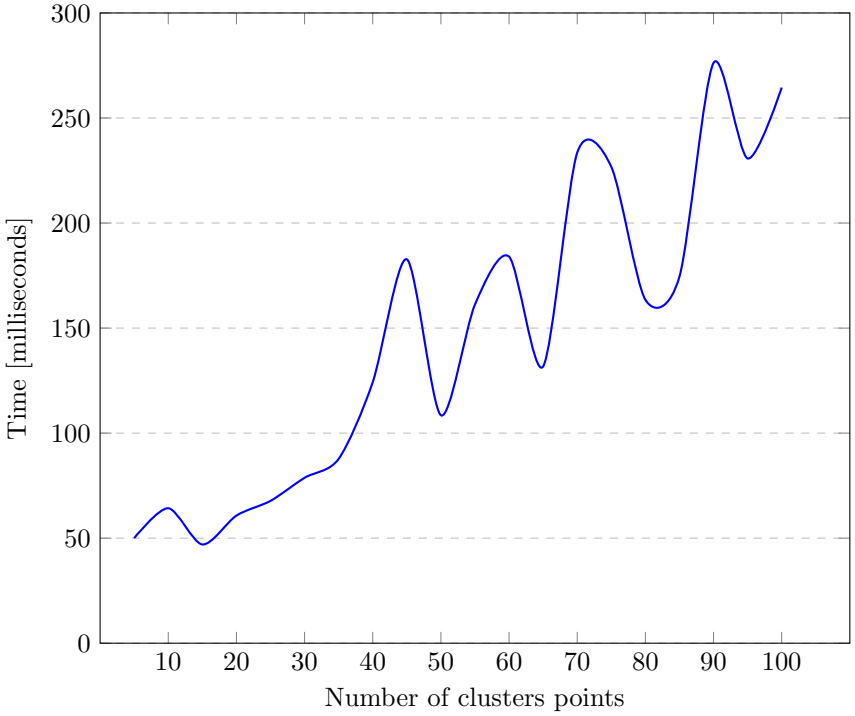
3.2.1 Sequential

Time dependence on the number of clusters with 30 000 data points



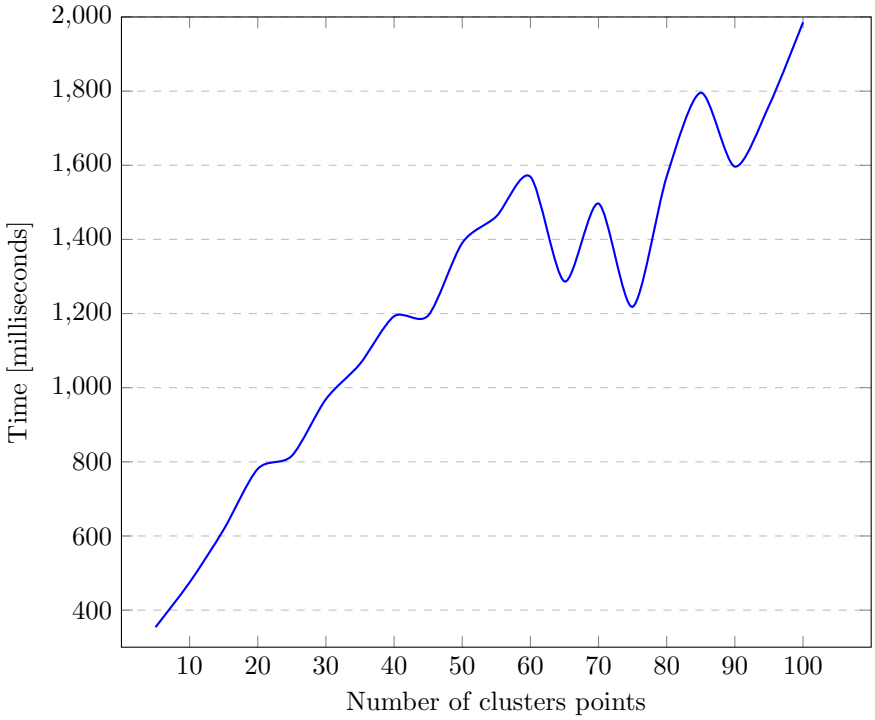
3.2.2 Parallel

Time dependence on the number of clusters with 30 000 data points



3.2.3 Distributed mode

Time dependence on the number of clusters with 30 000 data points



4 Conclusion

We pretty much got the result we were expecting. In the computations of smaller "volume" like the tests performed on a varying number of sites, sequential program performs better than the distributed one. The cost of the setup, packing and unpacking and message passing is greater than the speedup from concurrency. On the other hand, parallel program outperforms in this category already.

The true power of concurrent programs unlocks with computations of greater volume, like in the tests with the varying number of clusters. Here, concurrent programs outperformed the sequential one by a factor of cca. 200 and 7 respectively for the parallel and distributed versions.

Moreover, the parallel program constantly performs better than the distributed one. This is mostly due to the nature on the task at hand. Distributed programs are better suited for problems with more or less independent subtasks, since message passing can be costly. As we have seen, this is the case here. We have a lot of communication happening, because subtasks are dependent on each other's results .