# Distributed KMeans clustering

Emanuele Quaglio
Lorenzo Vigorelli

Paolo Lapo Cerni
Arman Singh Bains

11th July 2024


Università
degli Studi
di Padova

# The project



This project aims to adapt the well-known **KMeans clustering algorithm** to MapReduce-like architectures, exploiting the parallelization capabilities offered by distributed systems.
At the state of the art:

- KMeans consists of the **initialization** and the **Llyod iterations**. A proper initialization is crucial for obtaining good results.
- The **KMeans++** algorithm can obtain a set of initial centroids close to the optimal one but it's not easily parallelizable.
- Recently, **KMeans//** has been proposed to overcome this issue.

# The dataset



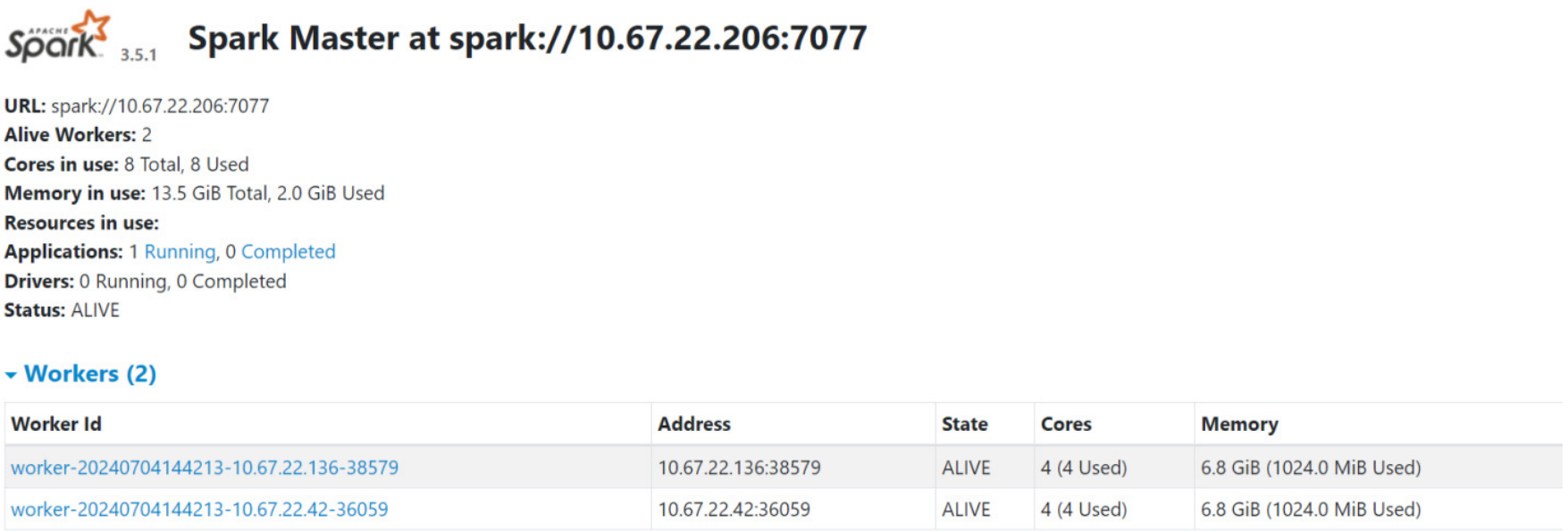
The **KDD Cup 1999** is a wide dataset for evaluating network intrusion detection systems. It contains:

- $\sim 500k$ records with attributes ($x$) and labels ($y$)
- 41 attributes for each record
- The labels correspond to the expected "*true*" clusters

# The hardware



We use **Spark** as the engine to distribute the analysis, exploiting **Cloud Veneto** computational resources.
Our cluster has a master node and two workers. Each machine has 4 CPUs with 1 thread per core and 4 sockets.



Spark 3.5.1 **Spark Master at spark://10.67.22.206:7077**

**URL:** spark://10.67.22.206:7077
**Alive Workers:** 2
**Cores in use:** 8 Total, 8 Used
**Memory in use:** 13.5 GiB Total, 2.0 GiB Used
**Resources in use:**
**Applications:** 1 Running, 0 Completed
**Drivers:** 0 Running, 0 Completed
**Status:** ALIVE

**Workers (2)**

| Worker Id | Address | State | Cores | Memory |
|---|---|---|---|---|
| worker-20240704144213-10.67.22.136-38579 | 10.67.22.136:38579 | ALIVE | 4 (4 Used) | 6.8 GiB (1024.0 MiB Used) |
| worker-20240704144213-10.67.22.42-36059 | 10.67.22.42:36059 | ALIVE | 4 (4 Used) | 6.8 GiB (1024.0 MiB Used) |

Figure: Spark WebUI

# Dataset exploration (1)

# k-Means // introduction

# Naive Initialization



Here we define a naive initialization function, where we uniformly sample centroids from points of the distributed dataset.

```
def naiveInitFromSet(Rdd, k, spark_seed=12345,logNaiveInit=None):
    """
    Uniform sampling of k points from Rdd
    Arguments:
    `Rdd`: see rdd structure;
    `k`: desired number of clusters;
    `spark_seed`: optional, seed for spark random sampling;
    `logNaiveInit`: optional, dictionary {'tTotal'} to store time info.
    Return:
    initial array (k, dim) of centroids.
    """
    t0 = time()
    # Sampling. Replacement is set to False to avoid coinciding centroids BUT no guarantees that in
    # the original dataset all points are distinct
    kSubset=Rdd.takeSample(False, k, seed=spark_seed)
    C_init=np.array([datum[1]['x'] for datum in kSubset])

    tEnd = time()

    if logNaiveInit is not None:
        logNaiveInit["tTotal"] = tEnd - t0

    return C_init
```

Alternatively, another function that uniformly samples from the vector space where the dataset is embedded can be used

# Parallel Initialization



The first part of K-Means // distributed initialization is made of: - sampling of an initial centroid calling ‘naiveInitFromSet‘; - computation of dataset-to-centroid distances by the ‘updateDistances‘ function; - computation of the clusterization cost with given centroid; - calculation of necessary number of K-Means // iterations as the logarithm of such cost.

```
def parallelInit(Rdd, k, l, logParallelInit=None):
    """
    Parallel initialization
    Arguments:
    `Rdd`: see rdd structure;
    `k`: desired number of clusters;
    `l`: coefficient to adjust sampling probability in order to obtain at least k centroids;
    `logParallelInit`: optional, dictionary {'CostsKmeans', 'tIterations', 'tTotal'} to store cost
    and time info.
    Return:
    Initial array (k, dim) of centroids.
    """
    t0 = time()

    # initialize C as a point in the dataset
    C=naiveInitFromSet(Rdd, 1)

    # associate each datum to the only centroid (computed before) and computed distances and cost
    Rdd=Rdd.map(lambda datum : (0, datum[1]))
    Rdd=updateDistances(Rdd, C).persist() ###

    my_cost=cost(Rdd)

    # number of iterations (log(cost))
    n_iterations=int(np.log(my_cost))
    if(n_iterations<1): n_iterations=1

    tSamples = []
    tCentroids = []
    CostInits = [my_cost]
```

Figure: Enter Caption

The second part of the initialization consists in iterating over: - the sampling of new points of the dataset to add to the centroids array, with probability proportional to their squared distance from their cluster centroid, calculated directly inside the ‘selectCluster‘ function; - the relabeling of the points according to nearest centroid, by the ‘selectCluster‘ function; - computation of total cost for probability normalization.

```
for _ in range(n_iterations):

    t1=time()
    # sample C' according to the probability
    C_prime=Rdd.filter(lambda datum : np.random.uniform()<l*datum[1]['d2']/my_cost)\
               .map(lambda datum : datum[1]['x'])\
               .collect()
    C_prime=np.array(C_prime)
    t2=time()

    # stack C and C', update distances, centroids, and cost
    if (C_prime.shape[0]>0):
        C=np.vstack((C, C_prime))

        # Rdd.unpersist() ###
        Rdd=Rdd.map(lambda datum: selectCluster(datum, C)).persist() ###

        my_cost=cost(Rdd)
    t3=time()

    tSample = t2 -t1
    tCentroid = t3 - t2
    tSamples.append(tSample)
    tCentroids.append(tCentroid)
    CostInits.append(my_cost)

# erase centroids sampled more than once
C=C.astype(float)
C=np.unique(C, axis=0)
Rdd=Rdd.map(lambda datum: selectCluster(datum, C))

# compute weights of centroids (sizes of each cluster) and put them in a list whose index is
# same centroid index as C
wx=Rdd.countByKey()
weights=np.zeros(len(C))
weights[[list(wx.keys())]]=[list(wx.values())]

#subselection of k centroids from C, using local Lloyds algorithm with k-means++ initialization
if C.shape[0]<=k:
    C_init=C
else:

    C_init=localLloyds(C, k, weights=weights, n_iterations=100)

tEnd = time()

if logParallelInit is not None:
    logParallelInit["tSamples"] = tSamples
    logParallelInit["tCentroids"] = tCentroids
    logParallelInit["CostInit"] = CostInits
    logParallelInit["tTotal"] = tEnd - t0

# Rdd.unpersist()
return C_init
```

# ‘selectCluster‘ function



```
def selectCluster(datum, C, updateDistances=True):
    """
    Associate datum to its centroid and optionally updates squared distance between them.
    Arguments:
    `datum`: see rdd format;
    `C`: array (k, len(datum[1]["x"]));
    `updateDistances`: if True, updates `datum[1]["d2"]` with squared distance between datum point
and closest centroid in C.
    Return:
    Updated datum.
    """
    distances = np.sum((datum[1]["x"] - C)**2, axis=1)
    print('distances: ',distances)
    clusterId = np.argmin(distances)
    if updateDistances is True:
        return (clusterId, {'x':datum[1]['x'], 'y':datum[1]['y'], 'd2':distances[clusterId]})
    else:
        return (clusterId, datum[1])
```

Figure: Enter Caption

For efficiency reason, the point-to-centroid distances can be updated directly by this function, with no need to invoke ‘updateDistances‘.

# ‘updateDistances‘ and ‘cost‘ functions



```
def updateDistances(Rdd, C):
    """
    Update Rdd with square distances from centroids, given Rdd with clusters already assigned to each
    point
    Arguments:
    `Rdd`: see rdd format;
    `C`: array of cluster centroids.
    Return:
    Updated rdd.
    """
    def datumUpdate(datum, C):
        '''
        Update a datum of the rdd with distance from assigned centroid
        '''
        d2=np.sum((datum[1]['x']-C[datum[0]])**2)
        #return datum
        return (datum[0], {"x": datum[1]["x"], "y": datum[1]["y"], "d2":d2})
    Rdd=Rdd.map(lambda datum:datumUpdate(datum, C))
    return Rdd
  def cost(Rdd):
    """
    Calculate global cost of clusterization, from an Rdd with distances from centroids already
    updated
    """
    my_cost=Rdd.map(lambda datum : datum[1]['d2'])\
               .reduce(lambda a,b: a+b)
    return my_cost
```

Figure: Enter Caption

```
def localPlusPlusInit(points, k):
    '''
    KMeans++ initialization.
    Arguments:
    `points`: array (n, dim) of points to be clustered;
    `k`: desired number of centroids.
    Returns:
    Initial array (k, dim) of centroids, k⩽n.
    '''
    # Sample one point uniformly from points array
    C=points[np.random.choice(points.shape[0])]
    C=C[np.newaxis, :]

    for _ in range(k):
        # Compute array (n,1) of probabilities associated to each point
        probs=np.min(np.sum((points[:,:,np.newaxis]-C.T[np.newaxis,:,:])**2, axis=1),
axis=1).flatten()
        # Normalize probability distribution
        probs=probs/np.sum(probs)

        # Draw one new centroid according to distrbution
        nextCentroid=points[np.random.choice(points.shape[0], p=probs)][np.newaxis,:]
        # Add centroid to array
        C=np.vstack((C, nextCentroid))
    return C
```

Figure: Enter Caption

# Local Lloyd



```python
def localLloyds(points, k, C_init=None, weights=None, n_iterations=100, logDict=None):
    """
    Local (non-distributed) Lloyds algorithm
    Arguments:
    `points`: array (n, dim) of points to cluster;
    `k`: number of desired clusters;
    `C_init`: optional, array (k, dim) of initial centroids
    `weights`: optional, weights for weighted average in centroid re-computing;
    `n_iterations`: optional, number of iteration in lloyds algorithm;
    `logDict`: optional, dictionary {'CostsKmeans', 'tIterations', 'tTotal'} to store cost and time
    info.
    Return:
    Array of expected centroids.
    """
    t0 = time()

    # Storing cost and time info
    my_kMeansCosts = []
    tIterations = []

    df=pd.DataFrame(points)

    # If weights not given, assume uniform weights for points
    if weights is None:
        weights=np.ones(shape=len(points))
    df['weights']=weights
    df['clusterId']=np.zeros(shape=len(points))

    # If no C_init, default to K-Means++ initialization
    if C_init is None:
        C=localPlusPlusInit(points, k)
    else:
        C=C_init
```

Figure: Local Lloyd first

```
    clusterId=np.argmin(np.sum((points[:,:,np.newaxis]-C.T[np.newaxis,:,:])**2, axis=1), axis=1)
    for iteration in range(n_iterations):
        t1=time()

        # Compute centroid given cluster
        df['clusterId']=clusterId
        C_df=df.groupby('clusterId')\
            .apply(weightedAverage)\
            .reset_index()

        # Compute cluster given centroid
        C_array=C_df[C_df.columns.difference(['weights',
                                              'clusterId'])].reset_index(drop=True).to_numpy()
        squared_distances=np.sum((points[:,:,np.newaxis]-C_array.T[np.newaxis,:,:])**2, axis=1)
        clusterId=np.argmin(squared_distances, axis=1)
        my_cost=sum(squared_distances[np.arange(len(squared_distances)), clusterId])

        my_kMeansCosts.append(my_cost)
        t2 = time()

        tIteration = t2 - t1
        tIterations.append(tIteration)

    tEnd = time()
    tTotal = tEnd - t0

    # Store cost and time info
    if logDict is not None:
        logDict["CostsKmeans"] = my_kMeansCosts
        logDict["tIterations"] = tIterations
        logDict["tTotal"] = tTotal

    return C_array
```

Figure: Enter Caption

# LLoyd's Algorithm



We defined a function to perform the LLoyd's parallelization

```
def kMeans(Rdd, C_init, maxIterations, logParallelKmeans=None):
    """
    Distributed (parallel) Lloyds algorithm
    Arguments:
    `Rdd`: see rdd format;
    `C_init`: array (k, dim) of initial centroids;
    `maxIterations`: max number of iterations;
    `logParallelKmeans`: optional, dictionary {'CostsKmeans', 'tIterations', 'tTotal'} to store cost
and time info.
    Return:
    Array of expected centroids.
    """

    t0 = time()

    # Storing cost and time info
    my_kMeansCosts = []
    tIterations = []
    C=C_init
```

Figure: Initial Lloyd's Algorithm implementation

```
    for t in range(maxIterations):
        t1 = time()
        RddCached = Rdd.map(lambda datum: selectCluster(datum, C)).persist() ###

        # Now we compute the new centroids by calculating the averages of points belonging to the
        # same cluster.
        C=updateCentroids(RddCached)
        my_cost = cost(RddCached)

        my_kMeansCosts.append(my_cost)
        t2 = time()

        tIteration = t2 - t1
        tIterations.append(tIteration)

        #RddCached.unpersist()

        # Break loop if convergence of cost is reached
        if (len(my_kMeansCosts) > 1) and (my_kMeansCosts[-1] > 0.999*my_kMeansCosts[-2]):
            break

    tEnd = time()
    tTotal = tEnd - t0

    # Store cost and time info in argument dictionary
    if logParallelKmeans is not None:
        logParallelKmeans["CostsKmeans"] = my_kMeansCosts
        logParallelKmeans["tIterations"] = tIterations
        logParallelKmeans["tTotal"] = tTotal

    return C
```

Figure: Completion of Lloyd's algorithm implementation

# K-means quality 1

- We need to evaluate the algorithm's by assessing the typical cluster distance using the **true labels** and comparing them with the **predicted labels performance**.

Figure 11 shows the population in each cluster, on the left for the predicted clusters, on the right for the expected ones

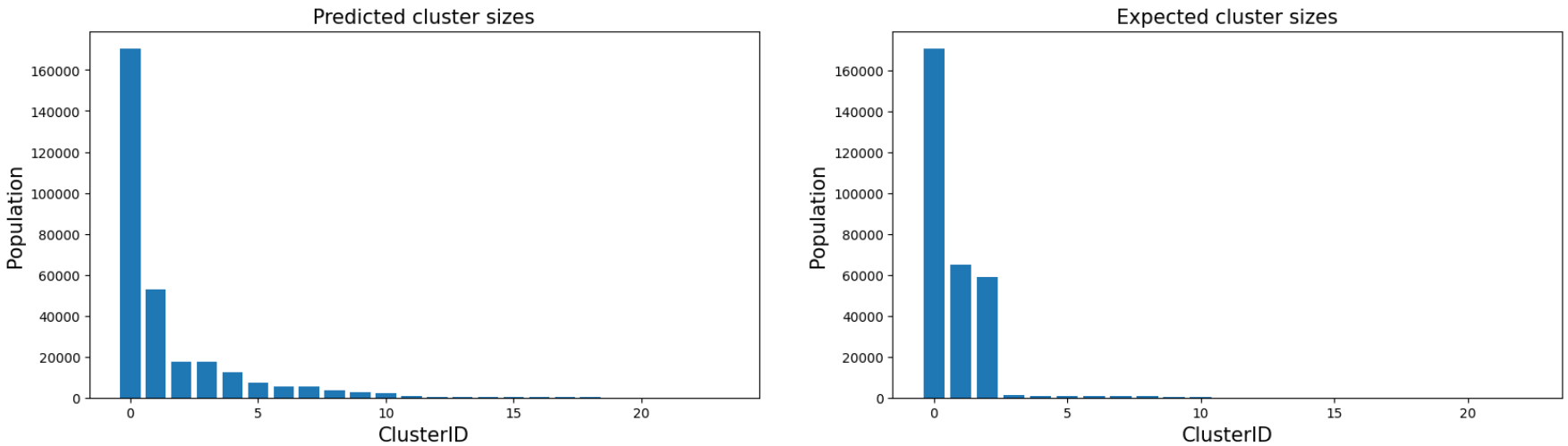
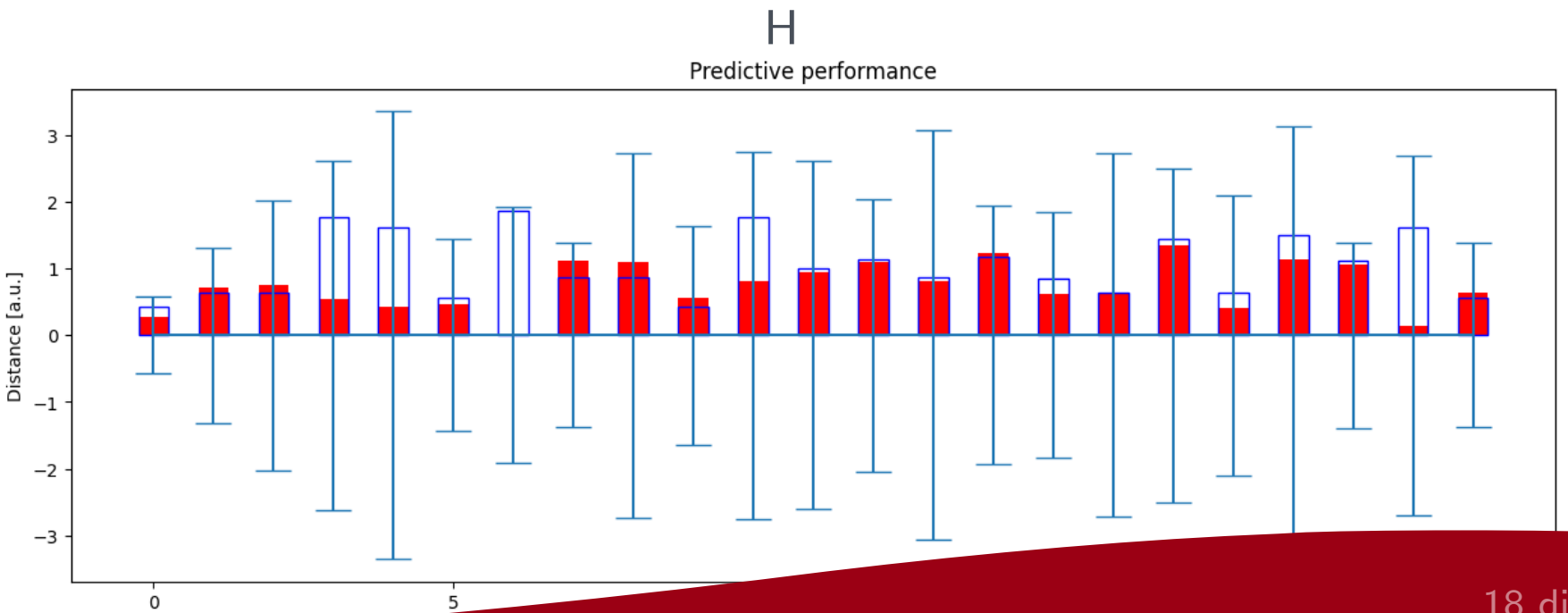



Figure: Predicted cluster sizes versus expected cluster sizes

# K-means quality

Figure 12 shows:

- how spread out the points in each cluster are around their respective centroids in the ground truth data (blue error bars)
- how far apart the clusters are from each other in the ground truth data (blue boxes)
- the error in the clustering algorithm's output compared to the ground truth (red bars)



# Memory optimization



All of the computation was done using a **subsample** of the data-set: 300k samples. We aren't able to explore bigger data-sets for **memory reasons**:

- The data processed in the Rdd was too much for the RAM of the virtual machine, causing **swapping**, this slowed the iteractions and make crash the program, due to the overload of the disk memory

Our solution to optimize, was to avoid using persists or using unpersist in strategical points of the functions, but this, as we did, caused the program to slow down a lot (4/5 times).

# Assessing K-means Performance



The most critical aspect of our study was evaluating the **performance** of the K-means algorithm initialization, focusing on both **time efficiency** and **cost**. We aimed to understand how initialization affects the algorithm's performance in different scenarios.

- **Time and Cost Analysis:**
  - We analyzed the algorithm's performance in terms of time and cost, emphasizing how initialization impacts these factors.
- **Effect of RDD Partitioning:**
  - We investigated how the number of RDD partitions influences the initialization time and overall performance.
- **Comparison with Random Initialization:**
  - We compared the performance of our initialization method with random initialization from points within the dataset.

# Methodology



- **Methodology:**
  For each run of the algorithm, including pre-processing, initialization, and Lloyd iterations, we logged detailed information about time efficiency and algorithm performance in a log file.
- **Structure of the Log File:**
  Each file, saved as a pickle, contains a dictionary with data from a scan over a predefined set of workers to compare parallel initialization and random initialization performances.
  The dictionary has a nested "dictionaries of dictionaries" structure to handle different run contexts in a unified and flexible manner.
- **Pros and Cons:**
  This structure allows for comprehensive data organization, though it introduces complexity in usage.

# Time Efficiency 1

We unpack the data to compare the **mean total execution time**. Results are shown in Figure 13.
This was done by **grouping** data from multiple runs, **number of partitions**, and the three **steps of each run**: pre-processing, initialization (either parallel or random), and Lloyd iterations.

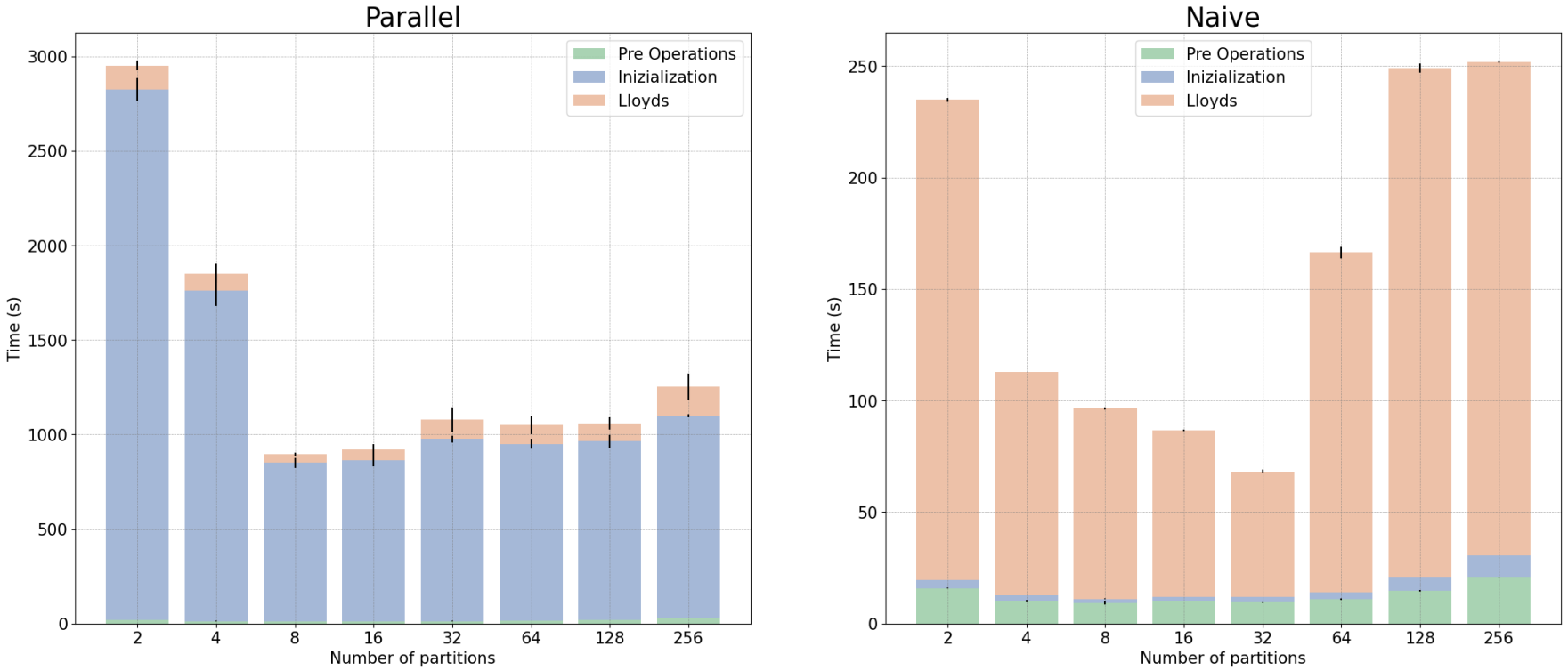



Figure: Bar plot for time effi

# Conclusions: Time and Cost Analysis



**Parallel initialization**

- Shows a significant reduction in total time as the number of partitions increases, particularly noticeable up to 8 partitions. Beyond this, the total time stabilizes but does not decrease significantly.
- The initialization time (blue) is the major time-consuming component initially, but reduces with increased partitions.

**Random initialization**

- Time for Lloyd's algorithm is considerably higher for smaller partitions, and though it decreases with more partitions, it starts increasing again after a certain point (64 partitions), suggesting inefficiency in large partitions
- Pre-operations and initialization times are minimal compared to Lloyd's time, indicating less influence on overall performance.

# Time Efficiency 2

We visually compare, in figure 14 the time spent based on the number of iterations and partitions used for **sampling** and **updating centroids**, specifically focusing on parallel initialization.

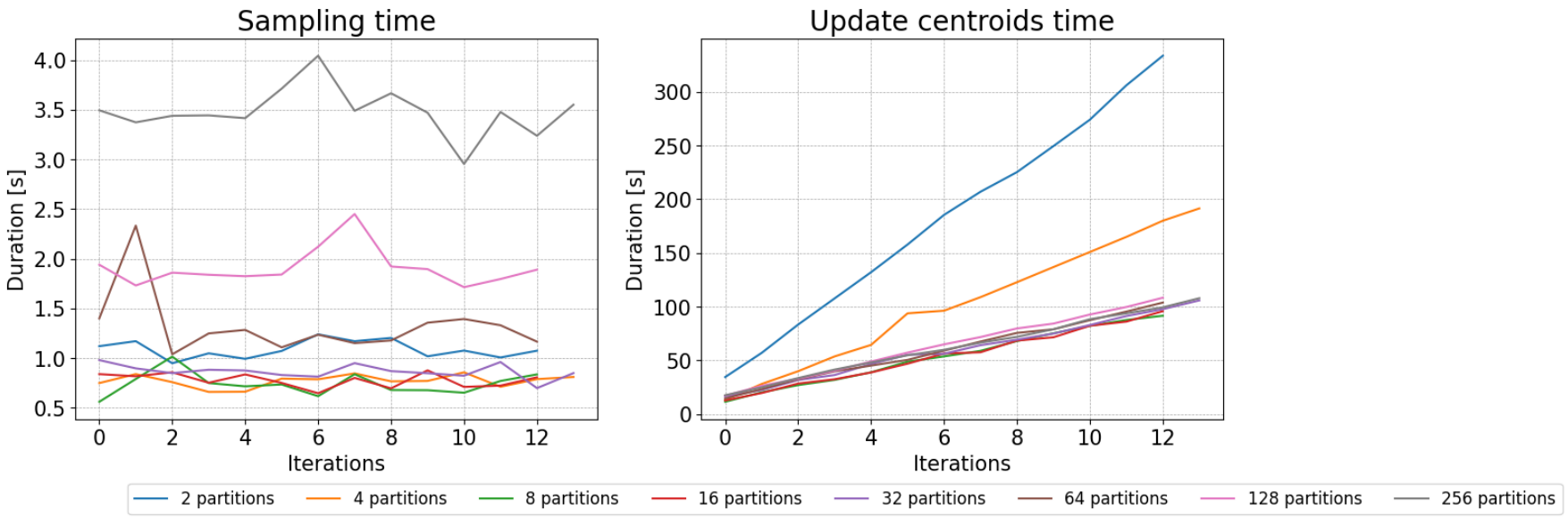



Figure: Plot of the main steps of parallel initialization

# Conclusions: Detailed Time Analysis



**Sampling time**

- Sampling time remains stable across iterations and partitions.
- Higher partition counts (128 and 256) tend to have slightly higher sampling times

**Updating centroids time**

- Updating centroids time increases linearly with iterations and partitions.
- Higher partitions show better scalability since they grow more gradually.

# Cost Analysis



We also analyze the dependency of the **cost function** on the number of iterations and partitions for **parallel** initialization, **Lloyd's**, and **random initialization**.
The cost function is defined as the sum of squared distances between each point and its nearest centroid:

$$\text{Cost} = \sum_{x} \min_{i=1,\dots,k} \|x - c_i\|^2$$

where $c_i$ is an element of the set of centroids and $x$ is a point in space.

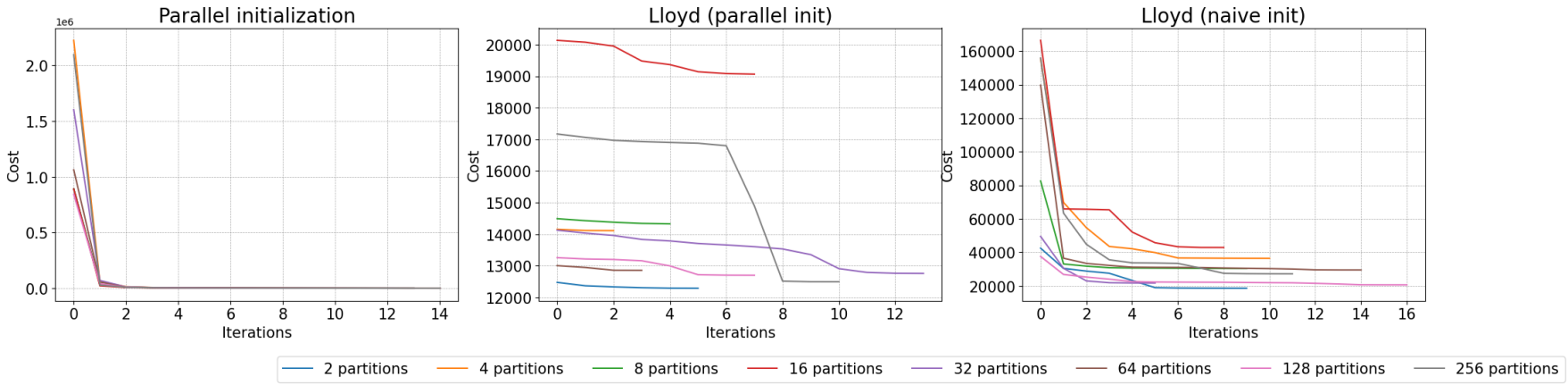



Figure: Plot of the cost for Parallel initialization, Lloyd (parallel init), and Lloyd (random init)

# Cost Analysis and Conclusions



**Parallel initialization**

- Quickly reduces and maintains low cost.

**Lloyd**

- Naive initialization shows a sharp decrease in cost in the first few iterations, stabilizing thereafter, while parallel initialization is at first very low, showing a slow decrease
- Naive initialization generally results in higher costs compared to parallel initialization, indicating less efficiency

# Final conclusions



**Conclusions**

- Parallel initialization outperforms random initialization.
- Proper partitioning enhances performance, the best results are obtained for a number of partition between 1 and 4 times the number of available cores.

**Future work**

- Since all of the computation was done with a subset of the dataset, improve the memory optimization, to be able to explore bigger datasets