ScalableKMeansPlusPlus

April 30, 2015

1 Scalable K-means++

1.1 Outline of Background

This project is an implementation based on Bahmani, Moseley, Vattani, Kumar and Vassilvitskii's paper *Scalable K Means++* in 2012.

1. ***K*-means** remains one of the most popular data processing algorithms. This algorithm has been used in many fields such us machine learning, pattern recognition and bioinformatics. However, the original k-means algorithm with random initialization has lots of weakness. For example.

   - A proper initialization is crucial for reciveing successful results.
   - For large dataset, it may take long time to achive convergent.

2. ***K*-means++** algorithm achived the goal of finding proper initialization, k-means++ has a deterministic initialization process, however, with downside of its inherent sequential nature, which limits its efficiency (*O(n)*) and applicability to big massive datasets.

3. ***k*-means||**, aka **Scalable *k*-means++**, *which proposed by this paper, oversamples by sampling each points independently with a larger probability, which is intuitively equivalent to updating the distribution much less frequently, with efficiency (* O(log n)*), which forms ***k*-means++** in both sequential and parallel settings.

In this project, I'll implement the *Scalable K Means++* algorithm, and compare it to the general *K Means* and *K Means++*.

1.2 Algorithm / Pseudocode

1.2.1 Notations

Let $X = \{x_1, ...x_n\}$ be the set of points in d-dimensional Euclidean space, and let k be a positive integer specifying the number of clusters. Let $||x_i$ - x_j $||$ denote the Euclidean distance between x_i and x_j. For a point x and a subset $Y \subseteq X$ of points, the distance is defined as $d(x, Y) = min_{y \in Y} ||x - y||$. For a subset $Y \subseteq X$ of points, let its centroid be given by

$$\text{centroid}(Y) = \frac{1}{|Y|} \sum_{y \in Y} y$$

Let $C = \{c_1, ...c_k\}$ be the set of points and let $Y \subseteq X$. We define the cost of Y with respect to C as

$$\phi_Y(C) = \sum_{y \in Y} d^2(y, C) = \sum_{y \in Y} \min_{i=1,...,k} ||y - c_i||^2$$

1.2.2 k-means++(k) initialization

1. $C \leftarrow$ sample a point uniformlt at random from X
2. **while** $|C| < k$ **do**

- Sample $x \in X$ with probability $\frac{d^2(x,C)}{\phi_X(C)}$
- $C \leftarrow C \cup \{x\}$

3. **end while**

1.2.3 k-means$||(k,l)$ initialization

1. $C \leftarrow$ sample a point uniformly at random from X
2. $\psi \leftarrow \phi_X(C)$
3. **for** $O(\log\psi)$ times **do**

- $C' \leftarrow$ sample each point $x \in X$ independently with probability $p_x = \frac{l \cdot d^2(x,C)}{\phi_X(C)}$
- $C \leftarrow C \cup C'$

4. **end for**
5. For $x \in C$, set w_x to be the number of points in X closer to x than any point in C
6. Recluster the weighted points in C into k clusters

1.3 Draft of unit test

Will verify the code correctness using following tests:

1. Test the Cost function using examples in terms of:

- non-negative
- data = c
- the size of c

2. Test Sampling Probability function using examples in terms of:

- 0 <= Pi <= 1
- sum = 1

3. Test K Means Plus Plus Parallel Probability function using examples in terms of:

- feasiblility
- _labels attributes

1.4 Data simulation

Simulating sample data from three bivariate normal distribution.

$$\begin{aligned}
\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} &\sim N_2\left(\begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 & 0.5 \\ 0 & 2 \end{pmatrix}\right) \\
\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} &\sim N_2\left(\begin{pmatrix} 3 \\ 7 \end{pmatrix}, \begin{pmatrix} 1 & 0.33 \\ 0.33 & 1 \end{pmatrix}\right) \\
\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} &\sim N_2\left(\begin{pmatrix} -8 \\ 2 \end{pmatrix}, \begin{pmatrix} 2 & 0.66 \\ 0.66 & 2 \end{pmatrix}\right)
\end{aligned}$$

```
In [48]: # Prepare
         #!/usr/bin/python
         from __future__ import division
         import os
         import sys
         import glob
         import random
         import sklearn
         import sklearn.cluster
         import numpy as np
         import pandas as pd
         import matplotlib.pyplot as plt
         from matplotlib.figure import Figure
         from matplotlib.axes import Subplot
         %matplotlib inline
         plt.style.use('ggplot')
```

```
In [49]: # Simulated "Real" Data Set
         class SimulatedData:
             def __init__(self, n, data):
                 self.data = data
                 self.n = n

             def DataSimulation(n):
                 mean1 = np.array([0, 1])
                 cov1 = np.array([[1, 0.5], [0.5, 2]])

                 mean2 = np.array([3, 7])
                 cov2 = np.array([[1, 0.33], [0.33, 1]])

                 mean3 = np.array([-8, 2])
                 cov3 = np.array([[2, 0.66], [0.66, 2]])

                 tmp = np.vstack((np.random.multivariate_normal(mean1, cov1, n),
                                  np.random.multivariate_normal(mean2, cov2, n),
                                  np.random.multivariate_normal(mean3, cov3, n)))
                 data = tmp[np.random.choice(range(3*n),size = 3*n, replace=False),]
                 return data

             def AsDataFrame(data):
                 df = pd.DataFrame(data,columns=["X","Y"])
                 return df
```

```
In [50]: data = SimulatedData.DataSimulation(5)
         data
```

```
Out[50]: array([[  3.27244498,   6.19079547],
                [  2.83259403,   5.31488359],
                [ -6.29887528,   1.15665735],
                [  2.26433884,   4.68468001],
                [  0.28221065,   1.48472561],
                [  4.05630379,   6.34592495],
                [  4.16797629,   7.93527882],
                [-10.70344788,   1.63873811],
                [ -1.34911855,  -2.49537396],
```

```
[ 1.52048191,  1.62214511],
[ -9.279794  ,  1.20963785],
[ 1.07200651,  2.15463174],
[ -9.58562466,  0.61057195],
[ -5.51352065,  3.68254202],
[ 0.84596127,  2.51050088]])
```

1.5 K-Means || Code

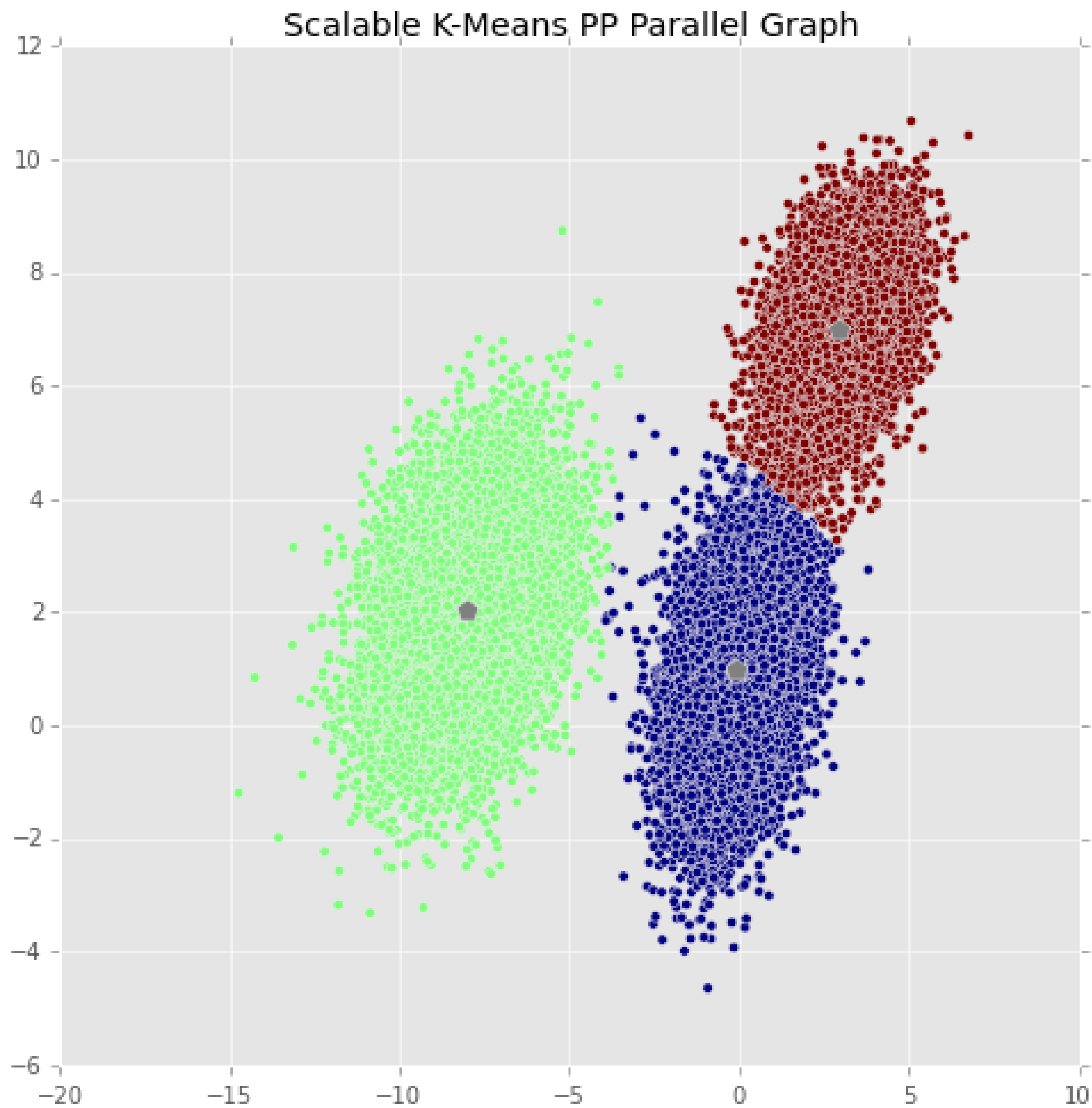
1.5.1 Navïe Version

```
In [51]: class ScalableKMeansPP:
             def __init__(self, data, k, l):
                 self.data = data
                 self.k = k
                 self.l = l

             def KMeansParallel(data, k, l):
                 N = data.__len__()
                 if k <= 0 or not(isinstance(k,int)) or l <= 0:
                     sys.exit()
                 # Then we start to Implement the algorithm
                 # 1. Sample one point uniformly at random from X
                 c = np.array(data[np.random.choice(range(N),1),])
                 # 2. To Cost function
                 phi = ScalableKMeansPP.CostFunction(c, data)
                 # 3. Looping
                 for i in range(np.ceil(np.log(phi)).astype(int)):
                     cPrime = data[ScalableKMeansPP.SamplingProbability(c,data,l) > np.random.uniform(si
                     c = np.concatenate((c, cPrime))
                 # End looping
                 # 7. For x in C, set w_x to be the number of pts closest to X
                 cMini = [np.argmin(np.sum((c-pts)**2,axis=1)) for pts in data];
                 closerPts = [cMini.count(i) for i in range(len(c))]
                 weight = closerPts/np.sum(closerPts)
                 # 8. Recluster the weighted points in C into k clusters
                 allC = data[np.random.choice(range(len(c)),size=1,p=weight),]
                 data_final = c
                 for i in range(k-1):
                     Probability = ScalableKMeansPP.SamplingProbability(allC,data_final,l) * weight
                     # choose next centroid
                     cPrimeFin = data[np.random.choice(range(len(c)), size=1, p=Probability/np.sum(Proba
                     allC = np.concatenate((allC,cPrimeFin))
                 KMeansPP = sklearn.cluster.KMeans(n_clusters=k, n_init=1, init=allC, max_iter=500, tol=
                 KMeansPP.fit(data);
                 return KMeansPP

             def SamplingProbability(c,data,l):
                 cost = ScalableKMeansPP.CostFunction(c,data)
                 return np.array([(min(np.sum((c-pts)**2,axis=1))) * l / cost for pts in data])

             def CostFunction(c,data):
                 return np.sum([min(np.sum((c-pts)**2,axis=1)) for pts in data])
```

```
In [52]: data = SimulatedData.DataSimulation(10000)
         ScalableKMeansPP.KMeansParallel(data,3,6)

Out[52]: KMeans(copy_x=True,
             init=array([[ 3.43704,  5.82661],
                [ 1.19989,  6.19787],
                [ 2.45681,  6.21065]]),
             max_iter=500, n_clusters=3, n_init=1, n_jobs=1,
             precompute_distances=True, random_state=None, tol=0.0001, verbose=0)
```

```
In [69]: k = 3
         start = timeit.default_timer()
         KMeansPP = ScalableKMeansPP.KMeansParallel(data=data, k=k, l=2*k); # paper suggesting using l=
         elapsed = timeit.default_timer() - start
         print(elapsed)
         df = SimulatedData.AsDataFrame(data)
         plt.figure(figsize=(8, 8));
         plt.scatter(df.X,df.Y,c=KMeansPP.labels_);
         plt.scatter(KMeansPP.cluster_centers_[:,0],KMeansPP.cluster_centers_[:,1], c='grey', s=100, mar
         plt.title("Scalable K-Means PP Parallel Graph");
         plt.show()
```

```
12.001063375006197
```



1.6 Algorithm Comparison

For algorithm comparison, I will using different sizes of same sample data to clustering using orginal k-means, k-means++, k-means||. Test their result and compare the efficiency.

We use "K-Means" and "K-Means++" methods from **sklearn.cluster.KMeans** package.

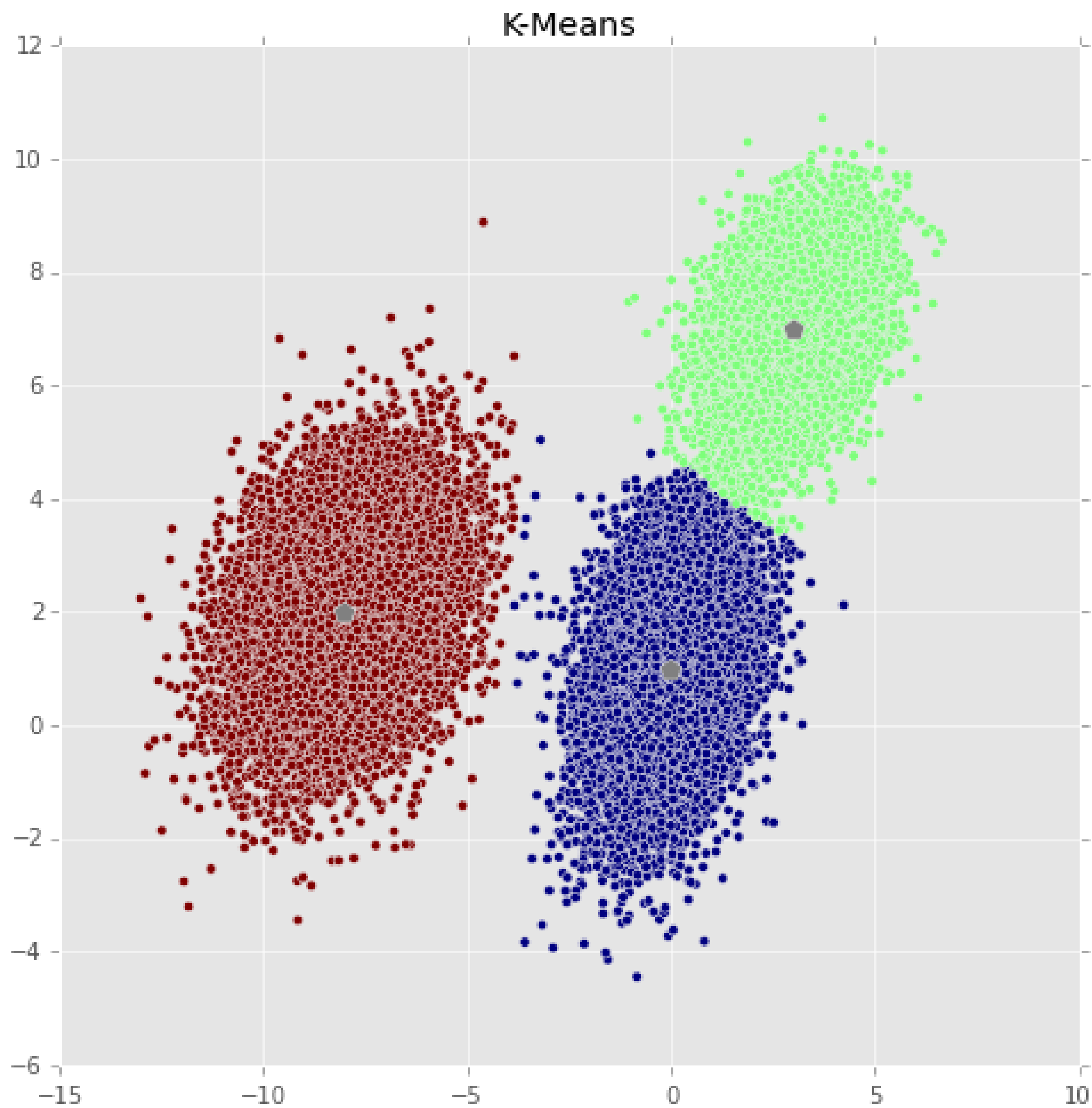
1.6.1 K-Means

In [75]:
```
import timeit
data1 = SimulatedData.DataSimulation(10000)
start = timeit.default_timer()
k = 3
KMeans = sklearn.cluster.KMeans(n_clusters=3, init='random', n_init=10, max_iter=500, tol=0.00
KMeans.fit(data1);
```

```
        elapsed = timeit.default_timer() - start
        print(elapsed)
```

0.12828452099347487

```
In [87]: KMeans.fit(data);
         df = SimulatedData.AsDataFrame(data)
         plt.figure(figsize=(8, 8));
         plt.scatter(df.X,df.Y,c=KMeans.labels_);
         plt.scatter(KMeans.cluster_centers_[:,0],KMeans.cluster_centers_[:,1], c='grey', s=100, marker
         plt.title("K-Means");
         plt.show()
```



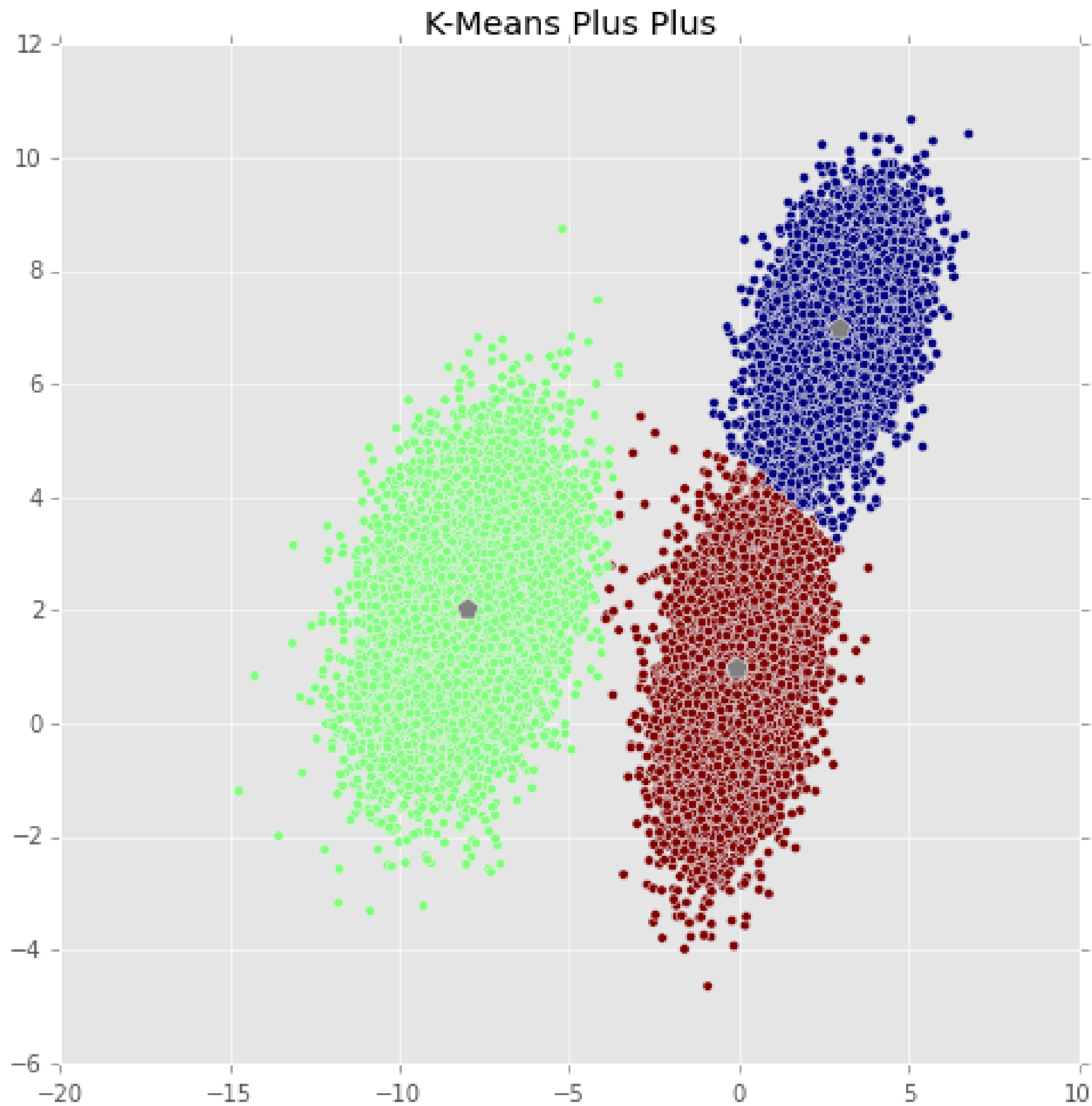
1.6.2 K-Means++

```
In [56]: data1 = SimulatedData.DataSimulation(10000)
         start = timeit.default_timer()
```

```
k = 3
KMeansPlusPlus = sklearn.cluster.KMeans(n_clusters=3, init='k-means++', n_init=10, max_iter=50
KMeansPlusPlus.fit(data1);
elapsed = timeit.default_timer() - start
print(elapsed)
```

```
0.08849192800698802
```

```
In [71]: KMeansPlusPlus = sklearn.cluster.KMeans(n_clusters=3, init='random', n_init=10, max_iter=500,
         KMeansPlusPlus.fit(data);
         df = SimulatedData.AsDataFrame(data)
         plt.figure(figsize=(8, 8));
         plt.scatter(df.X,df.Y,c=KMeansPlusPlus.labels_);
         plt.scatter(KMeansPlusPlus.cluster_centers_[:,0],KMeansPlusPlus.cluster_centers_[:,1], c='grey
         plt.title("K-Means Plus Plus");
         plt.show()
```



From the result of timeit, we can concluded that K-Means++ is much efficienct than the base K-Means. However, it seems that the navïe version of Scalable K-Means++ are slower than K-Means++, or even the base K-Means.

The reason would be, the package algorithms using parallel. I will try to optimization my code.

1.7 Optimization Strategies

1. Using alternative method to replace inefficent code in Python
2. Will try on large datasets, if still take too long. Could try to use other languages to write the looping part

1.8 Optimization: Vectorized Version

```
In [79]: class VectorizedScalableKMeansPP(ScalableKMeansPP):
             def __init__(self, data, k, l):
                 ScalableKMeansPP.__init__(self, data, k, l)

             def KMeansParallel(data, k, l):
                 N = data.__len__()
                 # 1. Sample one point uniformly at random from X
                 c = data[np.random.choice(range(data.shape[0]),1), :]
                 data__ = data[:,np.newaxis,:]
                 # 2. To Cost function
                 phi = ScalableKMeansPP.CostFunction(c, data)
                 # 3. Looping
                 for i in range(np.ceil(np.log(phi)).astype(int)):
                     d2 = (data__ - c) ** 2
                     distance = np.sum(d2, axis=2)
                     cMini = np.zeros(distance.shape)
                     cMini[range(distance.shape[0]), np.argmin(distance, axis=1)] = 1
                     min_dist = distance[cMini == 1]
                     phi = np.sum(min_dist)
                     for i, cPrime in enumerate(data):
                         Probability = l*min_dist[i]/phi
                         u = np.random.uniform(0,1)
                         if Probability >= u:
                             c = np.vstack([c, cPrime])
                 # End looping
                 # 7. For x in C, set w_x to be the number of pts closest to X
                 d2 = (data__ - c) ** 2
                 distance = np.sum(d2, axis=2)
                 cMini = np.zeros(distance.shape)
                 cMini[range(distance.shape[0]), np.argmin(distance, axis=1)] = 1
                 weight = np.array([np.count_nonzero(cMini[:, i]) for i in range(c.shape[0])]).reshape(
                 # 8. Recluster the weighted points in C into k clusters
                 allC = c[np.random.choice(range(c.shape[0]),1), :]
                 data_final = c
                 index = np.where(data_final==allC)[0]
                 data_final = np.delete(data_final,index[0],axis=0)
                 weight = np.delete(weight,index[0])
                 for i in range(k-1):
                     Probability = ScalableKMeansPP.SamplingProbability(allC,data_final,l) * weight
                     # choose next centroid
                     c = data_final[np.random.choice(range(data_final.shape[0]),size=1, p=Probability/np
```

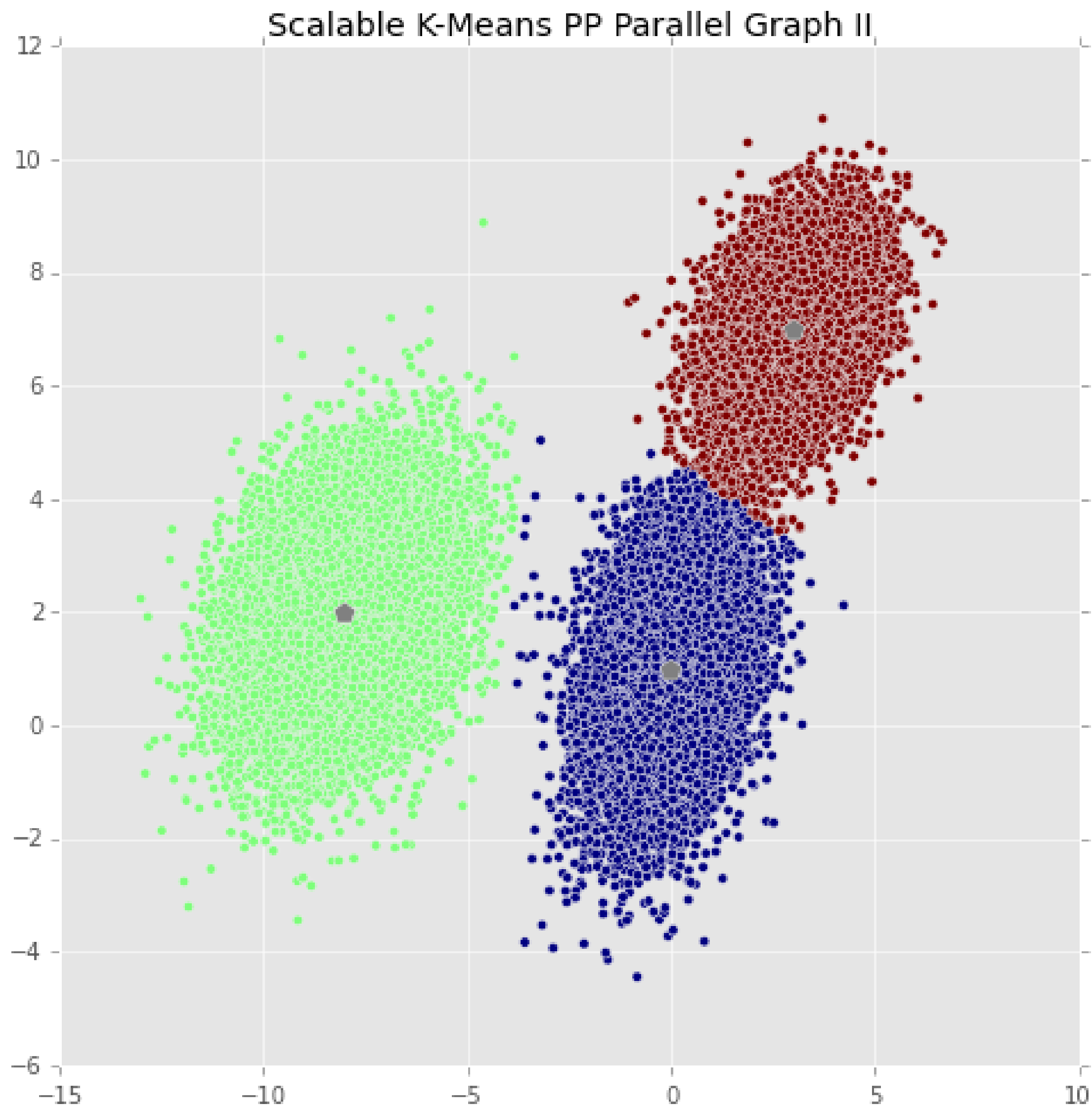
```
                index = np.where(data_final==c)[0]
                allC = np.vstack([allC, c])
                #Remove the selected center and its corresponding weight
                data_final = np.delete(data_final,index[0],axis=0)
                weight = np.delete(weight,index[0])
            vKMeansPP = sklearn.cluster.KMeans(n_clusters=k, n_init=1, init=allC, max_iter=500, tol
            vKMeansPP.fit(data)
            return vKMeansPP
```

In [80]:
```
data = SimulatedData.DataSimulation(10000)
VectorizedScalableKMeansPP.KMeansParallel(data,3,6)
```

Out[80]:
```
KMeans(copy_x=True,
    init=array([[-2.33086,  0.71013],
       [ 3.51431,  7.72725],
       [-8.37233,  3.60263]]),
    max_iter=500, n_clusters=3, n_init=1, n_jobs=1,
    precompute_distances=True, random_state=None, tol=0.0001, verbose=0)
```

In [85]:
```
start = timeit.default_timer()
KMeansPP2 = VectorizedScalableKMeansPP.KMeansParallel(data=data, k=k, l=2*k);
elapsed = timeit.default_timer() - start
print(elapsed)
df = SimulatedData.AsDataFrame(data)
plt.figure(figsize=(8, 8));
plt.scatter(df.X,df.Y,c=KMeansPP2.labels_);
plt.scatter(KMeansPP2.cluster_centers_[:,0],KMeansPP2.cluster_centers_[:,1], c='grey', s=100, m
plt.title("Scalable K-Means PP Parallel Graph II");
plt.show()
```

```
1.8935244909953326
```



1.9 Efficiency Comparsion: Profiling

In [ ]:
```
! pip install --pre line-profiler &> /dev/null
! pip install psutil &> /dev/null
! pip install memory_profiler &> /dev/null
```

In [18]:
```
%load_ext line_profiler
```

In [ ]:
```
%lprun -f ScalableKMeansPP.KMeansParallel ScalableKMeansPP.KMeansParallel(data = data, k=3, l =
```

In [ ]:
```
Timer unit: 1e-06 s

Total time: 13.3943 s
File: <ipython-input-125-d01bb3e3c9ef>
Function: KMeansParallel at line 7
```

```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     7                                           def KMeansParallel(data, k, l):
     8         1            4      4.0      0.0      N = data.__len__()
     9         1            2      2.0      0.0      if k <= 0 or not(isinstance(k,int)) or
    10                                                   sys.exit()
    11                                               # Then we start to Implement the algor
    12                                               # 1. Sample one point uniformly at ran
    13         1         3501   3501.0      0.0      c = np.array(data[np.random.choice(rang
    14                                               # 2. To Cost function
    15         1       270143 270143.0      2.0      phi = ScalableKMeansPP.CostFunction(c,
    16                                               # 3. Looping
    17        16           56      3.5      0.0      for i in range(np.ceil(np.log(phi)).as
    18        15     12559007 837267.1     93.8          cPrime = data[ScalableKMeansPP.Samp
    19        15          127      8.5      0.0          c = np.concatenate((c, cPrime))
    20                                               # End looping
    21                                               # 7. For x in C, set w_x to be the num
    22         1       369043 369043.0      2.8      cMini = [np.argmin(np.sum((c-pts)**2,a
    23         1       174695 174695.0      1.3      closerPts = [cMini.count(i) for i in ra
    24         1          114    114.0      0.0      weight = closerPts/np.sum(closerPts)
    25                                               # 8. Recluster the weighted points in
    26         1          112    112.0      0.0      allC = data[np.random.choice(range(len
    27         1            1      1.0      0.0      data_final = c
    28         3            5      1.7      0.0      for i in range(k-1):
    29         2         3734   1867.0      0.0          Probability = ScalableKMeansPP.Samp
    30                                                   # choose next centroid
    31         2          278    139.0      0.0          cPrimeFin = data[np.random.choice(
    32         2           13      6.5      0.0          allC = np.concatenate((allC,cPrime
    33         1           23     23.0      0.0      KMeansPP = sklearn.cluster.KMeans(n_cl
    34         1        13424  13424.0      0.1      KMeansPP.fit(data);
    35         1            2      2.0      0.0      return KMeansPP
```

In [82]:
```
%lprun -f VectorizedScalableKMeansPP.KMeansParallel VectorizedScalableKMeansPP.KMeansParallel(
```

In [ ]:
```
Timer unit: 1e-06 s

Total time: 3.66441 s
File: <ipython-input-114-00df5cba66a8>
Function: KMeansParallel at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           def KMeansParallel(data, k, l):
     6         1            7      7.0      0.0      N = data.__len__()
     7                                               # 1. Sample one point uniformly at ran
     8         1         3579   3579.0      0.1      c = data[np.random.choice(range(data.s
     9         1            4      4.0      0.0      data2 = data[:,np.newaxis,:]
    10                                               # 2. To Cost function
    11         1       264291 264291.0      7.2      phi = ScalableKMeansPP.CostFunction(c,
    12                                               # 3. Looping
    13        16           49      3.1      0.0      for i in range(np.ceil(np.log(phi)).as
    14        15       505753  33716.9     13.8          d2 = (data2 - c) ** 2
    15        15       254088  16939.2      6.9          distance = np.sum(d2, axis=2)
    16        15        36050   2403.3      1.0          cMini = np.zeros(distance.shape)
```

```
17        15      86425   5761.7      2.4              cMini[range(distance.shape[0]), np
18        15      33310   2220.7      0.9              min_dist = distance[cMini == 1]
19        15        796     53.1      0.0              phi = np.sum(min_dist)
20    450015     577435      1.3     15.8              for i, cPrime in enumerate(data):
21    450000     646465      1.4     17.6                  Probability = l*min_dist[i]/phi
22    450000     597961      1.3     16.3                  u = np.random.uniform(0,1)
23    450000     508532      1.1     13.9                  if Probability >= u:
24        75       2341     31.2      0.1                      c = np.vstack([c, cPrime])
25                                                 # End looping
26                                                 # 7. For x in C, set w_x to be the num
27         1      67089  67089.0      1.8          d2 = (data2 - c) ** 2
28         1      32498  32498.0      0.9          distance = np.sum(d2, axis=2)
29         1       5074   5074.0      0.1          cMini = np.zeros(distance.shape)
30         1       6751   6751.0      0.2          cMini[range(distance.shape[0]), np.argm
31         1      15825  15825.0      0.4          weight = np.array([np.count_nonzero(cMi
32                                                 # 8. Recluster the weighted points in
33         1         87     87.0      0.0          allC = c[np.random.choice(range(c.shape
34         1          2      2.0      0.0          data_final = c
35         1         16     16.0      0.0          index = np.where(data_final==allC)[0]
36         1         48     48.0      0.0          data_final = np.delete(data_final,index
37         1         32     32.0      0.0          weight = np.delete(weight,index[0])
38         3          4      1.3      0.0          for i in range(k-1):
39         2       2686   1343.0      0.1              Probability = ScalableKMeansPP.Samp
40                                                     # choose next centroid
41         2        140     70.0      0.0              c = data_final[np.random.choice(ran
42         2         17      8.5      0.0              index = np.where(data_final==c)[0]
43         2         43     21.5      0.0              allC = np.vstack([allC, c])
44                                                     #Remove the selected center and its
45         2         62     31.0      0.0              data_final = np.delete(data_final,i
46         2         55     27.5      0.0              weight = np.delete(weight,index[0])
47         1         20     20.0      0.0          vKMeansPP = sklearn.cluster.KMeans(n_cl
48         1      16874  16874.0      0.5          vKMeansPP.fit(data)
49         1          3      3.0      0.0          return vKMeansPP
```

By profiling, it is clearlt that Vectored KMeansParallel function is significantly faster than the navie KMeansParallel function. (move than three times faster) The total time was improved from 13.4s to 3.66s.