Convex Hull Using K-Means Clustering

Parallel Implementation

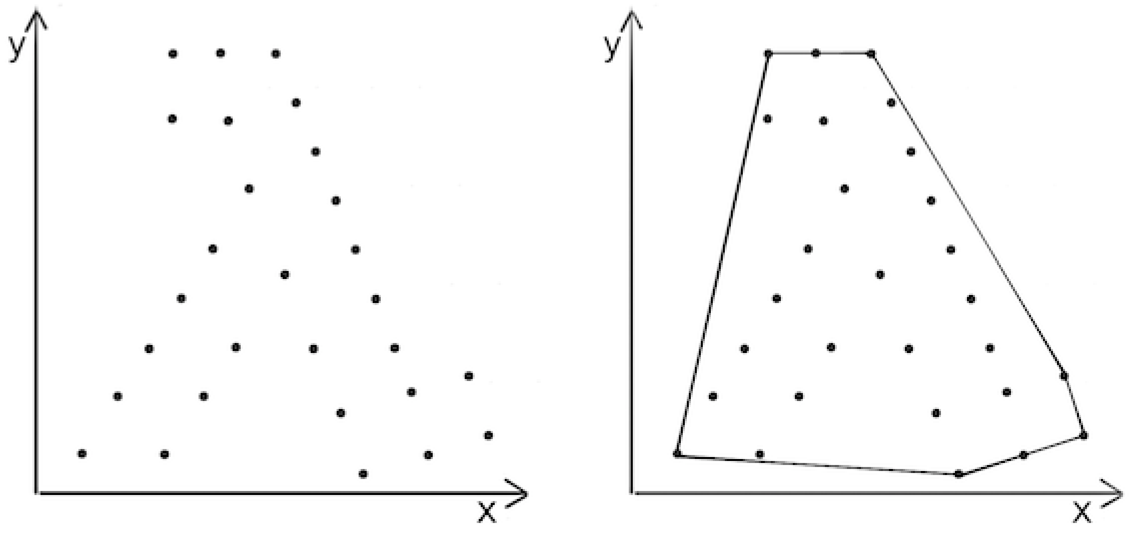
2019045 Bhargav Gohil
2019059 Gargi Chaurasia
2019221 Ayush Dubey
2019273 Kartik Mohan Garg

Submitted To: Dr. Manish Kumar Bajpai

Date: Oct 30, 2021

Problem Statement

We need to compute the convex hull based on a set of points in a two-dimensional space. Given a set S of n points in the plane, the convex hull of S is the smallest convex region containing all points in S.
The Basic idea is to improve runtime of the algorithm using K-means clustering technique.



Reference Research Paper

[Convex Hull Using K-Means Clustering in Hybrid(MPI/OpenMP) Environment]

Introduction

Convex Hull

In geometry, the convex hull or convex envelope or convex closure of a shape is the smallest convex set that contains it. The convex hull may be defined either as the intersection of all convex sets containing a given subset of a Euclidean space, or equivalently as the set of all convex combinations of points in the subset. For a bounded subset of the plane, the convex hull may be visualized as the shape enclosed by a rubber band stretched around the subset.

Significance in Research

One of the reasons for the choice of the convex hull is that, in addition hull of S is the smallest convex region containing all points to be considered a main topic in Computational Geometry, convex hulls have attracted the interest of researchers from a wide range of areas. Because of the complexity among others, they have been proved to be useful in applications such as pattern recognition, including algorithms for detecting human faces, reading a license plate or analyzing soil particles; computer graphics, computerized tomography, collision detection, prediction of chemical equilibrium or ecology.

Finding Convex Hull

The algorithmic problems of finding the convex hull of a finite set of points in the plane or other low-dimensional Euclidean spaces, and its dual problem of intersecting half-spaces, are fundamental problems of computational geometry. They can be solved in time $O(n\, log\, n)$ for two or three dimensional point sets, and in time matching the worst-case output complexity given by the upper bound theorem in higher dimensions.

If the number of points increases, the time required to create a Convex hull also increases. To reduce this time, the problem must be broken into multiple threads or processes. Given a set S of n points in the plane, the convex hull of S is the smallest convex region containing all points in S. The convex hull (S) is considered as an ordered list.

Different Algorithms

Many different design approaches lead to optimal (or expected optimal) solutions. These include divide and conquer, sweep line, incremental randomized constructions,Quick Hull, Graham scan and many others. This fact, together with the possibility of optimizing, its execution time and expected memory requirement, makes the convex hull an excellent benchmark for testing different architectures and programming models.

The divide and conquer algorithm is an elegant method for computing the Convex hull of a set of points based on this widely-known algorithm design technique. The main goal of opting for a parallel program is to utilize the multicore resource variable in common for improving the performance of the convex hull algorithm.

Terminologies

Quickhull Algorithm

Quickhull is a method of computing the convex hull of a finite set of points in n-dimensional space. It uses a divide and conquer approach similar to that of quicksort, from which its name derives. Its worst case complexity for 2-dimensional and 3-dimensional space is considered to be $O(n \, log \, r)$, where n is the number of input points and r is the number of processed points.

Clustering

It can be defined as the task of identifying subgroups in the data such that data points in the same subgroup (cluster) are very similar while data points in different clusters are very different. In other words, we try to find homogeneous subgroups within the data such that data points in each cluster are as similar as possible according to a similarity measure such as euclidean-based distance or correlation-based distance.

K Means Algorithm

K Means is one of the most used algorithms for finding clusters. It is an algorithm that tries to partition the dataset into *K* pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible.

Parallelising K-means clustering

For the problem of finding Convex Hull using K-means clustering, the section of K-means clustering is parallelism in this report.
K-means clustering is a hard clustering algorithm which means that each datapoint is assigned to one cluster (rather than multiple clusters with different probabilities). The algorithm starts with random cluster assignments and iterates between two steps:

- Assigning data points to clusters based on the closest centroid (by some distance metric)
- Updating the centroids based on the new cluster assignments from the previous step

Eventually the cluster assignments converge giving the final result. In this CUDA implementation each of the two steps will be performed in parallel.

K-Means Algorithm Based on Shared Memory Multiprocessors

The parallel k-means algorithm can be divided into 4 steps. These steps are initialization, distance calculation, centroid recalculation, and convergence. The first step of k-means is initialization of centroids. Potentially, this first step is not parallelizable; each centroid must be assigned globally. However, by exploiting the shared memory architecture, each processor could be assigned to the task of selecting C/P centroids. At the end of this operation a set of globally accessible centroids will be decided among processors.

Implementation

Phase II: Parallel Implementation **Date: 30.10.2021**

➔ Assigning Data points to clusters
The first step is assigning data points to their nearest centroid. This step is not difficult to parallelize because the distance computations can be performed independently for each datapoint.

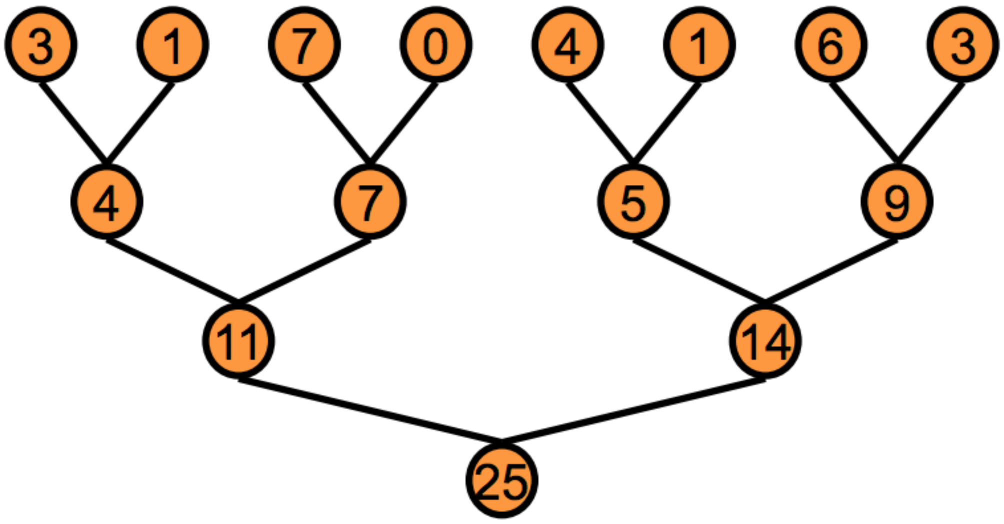
```
inline __global__ void kMeansClusterAssignment(int N, int* d_datapoint_x,
int* d_datapoint_y, int* d_clust_assn, int* d_centroids_x, int*
d_centroids_y) {
      //get idx for this datapoint
      const int idx = blockIdx.x * blockDim.x + threadIdx.x;

      //bounds check
      if (idx >= N) return;

      //find the closest centroid to this datapoint
      float min_dist = INFINITY;
      int closest_centroid = 0;

      for (int c = 0; c < K; ++c) {
            float dist = distance(d_datapoint_x[idx], d_datapoint_y[idx],
d_centroids_x[c], d_centroids_y[c]);
            if (dist < min_dist) {
                  min_dist = dist;
                  closest_centroid = c;
            }
      }
      //assign closest cluster id for this datapoint/thread
      d_clust_assn[idx] = closest_centroid;
}
```

The above code is actually very similar to the serial case. Here the thread index `idx` corresponds to the index of the datapoint. The rest of the code is pretty straight-forward. This distance between the datapoint and each centroid is computed and the centroid that is closest is then assigned to that datapoint. One drawback of this code is that the centroids and data points are read from global memory which is somewhat slow.

The distance function used here is simply the Euclidean distance for the 2-d case.

```
inline __device__ float distance(int x1, int y1, int x2, int y2) {
      return sqrt((float)(x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
}
```

- ➔ Updating Centroids
  The next step is to recompute the centroids given the cluster assignments computed in the previous step. This is much more tricky to parallelize since the centroid computations depend on all of the other data points in its cluster. However, operations that rely on distributed datasets to compute a single output value can still be parallelized and are called reductions. A sum reduction is shown in the diagram below.



The general procedure is to partition the input array and perform the sum on each partition in parallel then merge the partitions and repeat the process until all partitions have been merged and you are left with the final sum value. This parallelization allows for logarithmic complexity rather than linear as with the serial case.

The centroid recomputation can also be thought of as a reduction operation and the code is shown below.

```
inline __global__ void kMeansCentroidUpdate(int N, int* d_datapoints_x,
int* d_datapoints_y, int* d_clust_assn, int* d_centroids_x, int*
d_centroids_y, int* d_clust_sizes) {
      //get idx of thread at grid level
      const int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
    //bounds check
    if (idx >= N) return;

    //get idx of thread at the block level
    const int s_idx = threadIdx.x;

    //put the data points and corresponding cluster assignments
    //in shared memory so that they can be summed by thread 0 later
    __shared__ int s_datapoints_x[TPB];
    __shared__ int s_datapoints_y[TPB];

    s_datapoints_x[s_idx] = d_datapoints_x[idx];
    s_datapoints_y[s_idx] = d_datapoints_y[idx];


    __shared__ int s_clust_assn[TPB];
    s_clust_assn[s_idx] = d_clust_assn[idx];

    __syncthreads();

    //it is the thread with idx 0 (in each block) that sums up all
    //the values within the shared array for the block it is in
    if (s_idx == 0) {
          int b_clust_datapoint_sums_x[K] = { 0 };
          int b_clust_datapoint_sums_y[K] = { 0 };
          int b_clust_sizes[K] = { 0 };
          for (int j = 0; j < blockDim.x; ++j) {
                int clust_id = s_clust_assn[j];
                b_clust_datapoint_sums_x[clust_id] += s_datapoints_x[j];
                b_clust_datapoint_sums_y[clust_id] += s_datapoints_y[j];
                b_clust_sizes[clust_id] += 1;
          }

          //Now we add the sums to the global centroids and
          //add the counts to the global counts.
          for (int z = 0; z < K; ++z) {
                atomicAdd(&d_centroids_x[z],b_clust_datapoint_sums_x[z]);
                atomicAdd(&d_centroids_y[z],b_clust_datapoint_sums_y[z]);
                atomicAdd(&d_clust_sizes[z], b_clust_sizes[z]);
          }
    }

    __syncthreads();
```

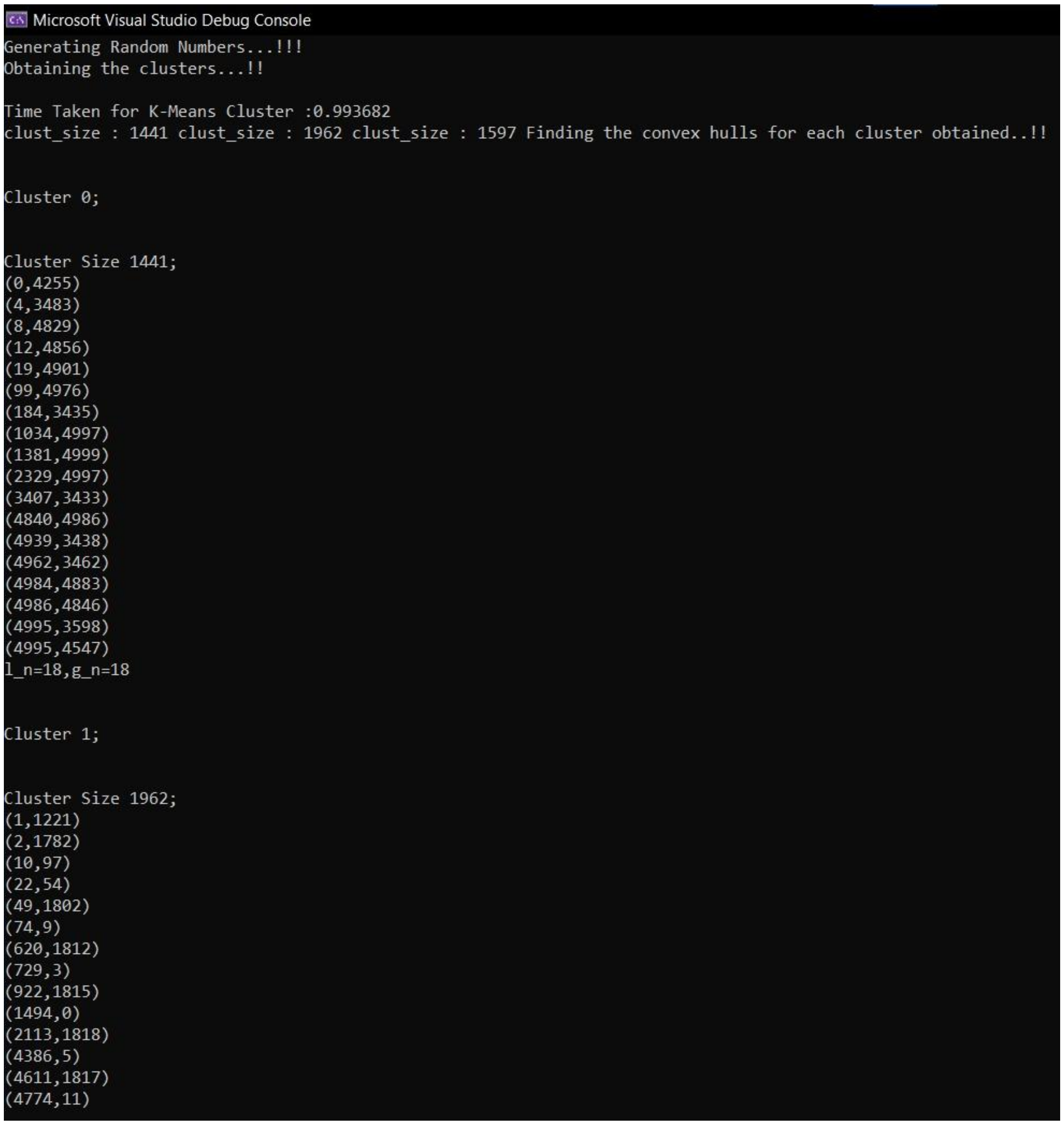
```
      //currently centroids are just sums, so divide by size to get
      //actual centroids
      if (idx < K) {
            d_centroids_x[idx] = d_centroids_x[idx] / d_clust_sizes[idx];
            d_centroids_y[idx] = d_centroids_y[idx] / d_clust_sizes[idx];
      }
}
```

The next bit of code is the partition step of the reduction where the global datapoint (`d_datapoints_x` and `d_datapoints_y`) and cluster assignment (`d_clust_assn`) variables are brought into shared memory for the block that thread `idx` is in.

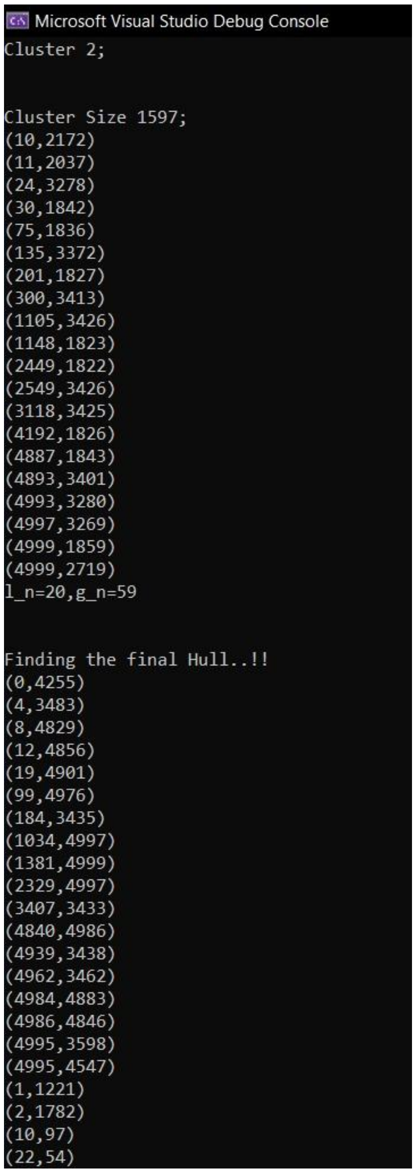
```
__shared__ int s_datapoints_x[TPB];
      __shared__ int s_datapoints_y[TPB];

      s_datapoints_x[s_idx] = d_datapoints_x[idx];
      s_datapoints_y[s_idx] = d_datapoints_y[idx];

      __shared__ int s_clust_assn[TPB];
      s_clust_assn[s_idx] = d_clust_assn[idx];
```

Results

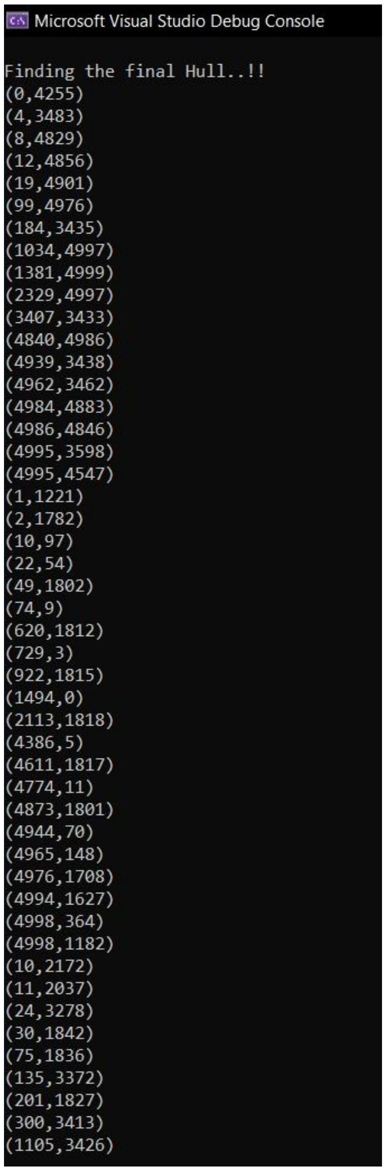
For K=3 and n=1500, we get the following output.


Microsoft Visual Studio Debug Console

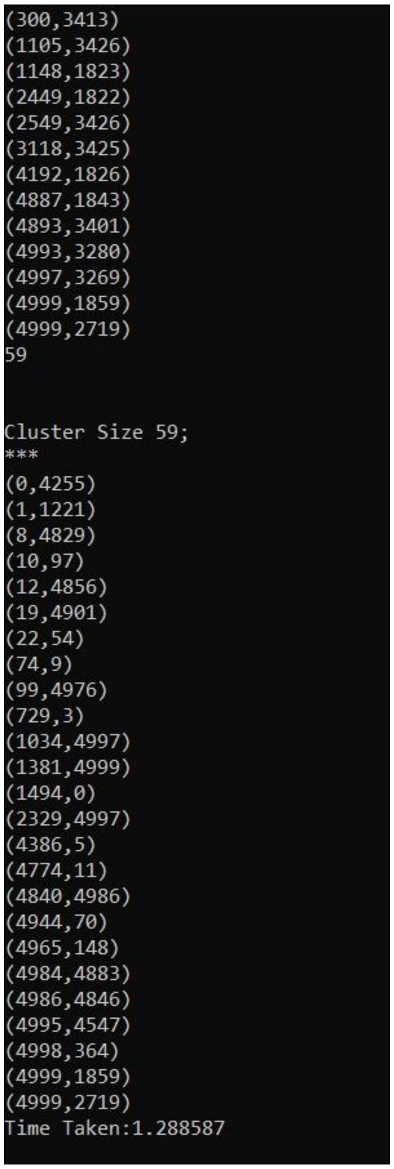
```
Generating Random Numbers...!!!
Obtaining the clusters...!!

Time Taken for K-Means Cluster :0.993682
clust_size : 1441 clust_size : 1962 clust_size : 1597 Finding the convex hulls for each cluster obtained..!!


Cluster 0;


Cluster Size 1441;
(0,4255)
(4,3483)
(8,4829)
(12,4856)
(19,4901)
(99,4976)
(184,3435)
(1034,4997)
(1381,4999)
(2329,4997)
(3407,3433)
(4840,4986)
(4939,3438)
(4962,3462)
(4984,4883)
(4986,4846)
(4995,3598)
(4995,4547)
l_n=18,g_n=18


Cluster 1;


Cluster Size 1962;
(1,1221)
(2,1782)
(10,97)
(22,54)
(49,1802)
(74,9)
(620,1812)
(729,3)
(922,1815)
(1494,0)
(2113,1818)
(4386,5)
(4611,1817)
(4774,11)
```



```
Cluster 2;


Cluster Size 1597;
(10,2172)
(11,2037)
(24,3278)
(30,1842)
(75,1836)
(135,3372)
(201,1827)
(300,3413)
(1105,3426)
(1148,1823)
(2449,1822)
(2549,3426)
(3118,3425)
(4192,1826)
(4887,1843)
(4893,3401)
(4993,3280)
(4997,3269)
(4999,1859)
(4999,2719)
l_n=20,g_n=59


Finding the final Hull..!!
(0,4255)
(4,3483)
(8,4829)
(12,4856)
(19,4901)
(99,4976)
(184,3435)
(1034,4997)
(1381,4999)
(2329,4997)
(3407,3433)
(4840,4986)
(4939,3438)
(4962,3462)
(4984,4883)
(4986,4846)
(4995,3598)
(4995,4547)
(1,1221)
(2,1782)
(10,97)
(22,54)
```



```

Finding the final Hull..!!
(0,4255)
(4,3483)
(8,4829)
(12,4856)
(19,4901)
(99,4976)
(184,3435)
(1034,4997)
(1381,4999)
(2329,4997)
(3407,3433)
(4840,4986)
(4939,3438)
(4962,3462)
(4984,4883)
(4986,4846)
(4995,3598)
(4995,4547)
(1,1221)
(2,1782)
(10,97)
(22,54)
(49,1802)
(74,9)
(620,1812)
(729,3)
(922,1815)
(1494,0)
(2113,1818)
(4386,5)
(4611,1817)
(4774,11)
(4873,1801)
(4944,70)
(4965,148)
(4976,1708)
(4994,1627)
(4998,364)
(4998,1182)
(10,2172)
(11,2037)
(24,3278)
(30,1842)
(75,1836)
(135,3372)
(201,1827)
(300,3413)
(1105,3426)
```



```
(300,3413)
(1105,3426)
(1148,1823)
(2449,1822)
(2549,3426)
(3118,3425)
(4192,1826)
(4887,1843)
(4893,3401)
(4993,3280)
(4997,3269)
(4999,1859)
(4999,2719)
59


Cluster Size 59;
***
(0,4255)
(1,1221)
(8,4829)
(10,97)
(12,4856)
(19,4901)
(22,54)
(74,9)
(99,4976)
(729,3)
(1034,4997)
(1381,4999)
(1494,0)
(2329,4997)
(4386,5)
(4774,11)
(4840,4986)
(4944,70)
(4965,148)
(4984,4883)
(4986,4846)
(4995,4547)
(4998,364)
(4999,1859)
(4999,2719)
Time Taken:1.288587
```