Convex Hull Using K-Means Clustering

Serial Implementation

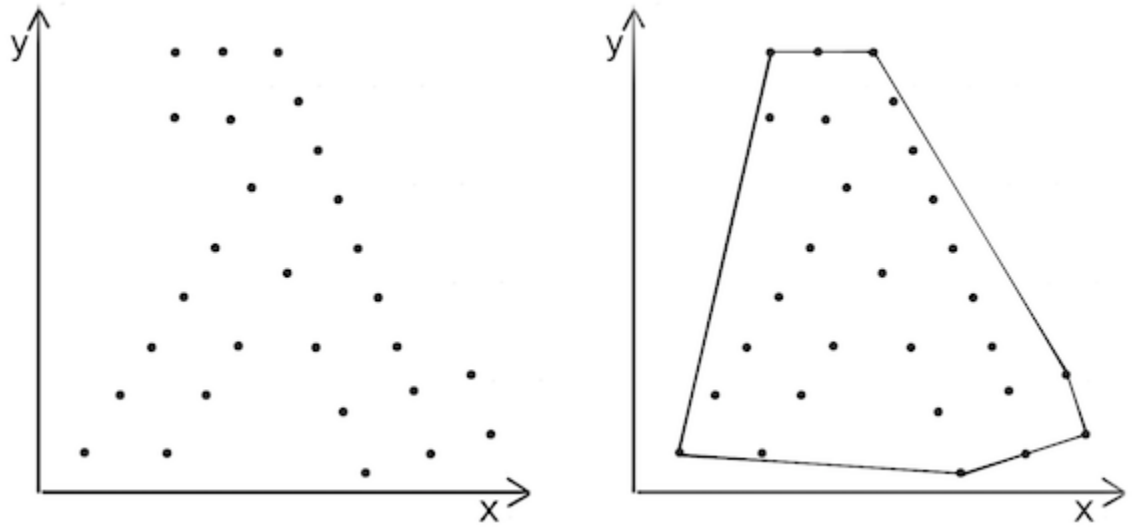
2019045 Bhargav Gohil
2019059 Gargi Chaurasia
2019221 Ayush Dubey
2019273 Kartik Mohan Garg

Submitted To: Dr. Manish Kumar Bajpai

Date: Sep 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.

Implementation

Phase I: Serial Code Implementation **Date: 30.09.2021**

To generate the set of n points,

```
for(int i = 0; i < n; i++){
      X[i] = rand() % 100;
      Y[i] = rand() % 100;
}
```

Now, generate clusters of the set of n points using K-Means algorithm

The way K-Means algorithm work:

1. Specify number of clusters K.
2. Initialize centroids by first shuffling the dataset and then randomly selecting K data points for the centroids without replacement.
3. Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.
    - Compute the sum of the squared distance between data points and all centroids.
    - Assign each data point to the closest cluster (centroid).
    - Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.

```
/**
 * @description Generate all the clusters
 * @param [in] n number of points in a 2D plane
 * @param k number of clusters
 * @param x,y array pointers storing x and y points
 * @param kc cluster array
 * @return null
 */
void find_clusters(int n,int k,int *x,int *y,int *kc){
    int min=999999,max_it=50,it=0;
```

```
    int xcnt=0,cnt=0;
    int cenx[n],ceny[n],kc1[n];
    double d[n],euc_distance[k][n];

    srand(time(0));
    for(int i = 0; i < n; i++){
        kc1[i]=rand()%k;
        kc[i]=rand()%k;
        cenx[i]=x[i];
        ceny[i]=y[i];
    }

    do{
        // "Calculating the euclidean distances"
        for(int m=0; m < k; m++) { // m is for cluster
            for(int j = 0; j < n; j++) { //j is for object
                d[j]=(((cenx[m]-x[j])*(cenx[m]-x[j]))
                +((ceny[m]-y[j]) * (ceny[m]-y[j])));
                euc_distance[m][j] = sqrt(d[j]);
            }
        }
        // Now,comparing if kc1 is equal to kc.
        int l = 0;
        for(int j = 0; j < n; j++){
            if(kc1[j] == kc[j]) { //checking each element
                l++;
                // It checks if all the elements are equal.
            }
        }
        if(l == n) return; // if yes then go outside this whole do-while
        else { // or just put the values of kc to kc1.
            for(int j = 0; j < n; j++) kc1[j] = kc[j];
        }
        //"Checking which data object is nearer to which cluster and
creating cluster."

        for(int j = 0; j < n; j++){ // data objects
            for(int m = 0; m < k; m++){ // cluster number
                if(euc_distance[m][j] < min) {
```

```
                    min = euc_distance[m][j];
                    kc[j] = m;
                }
            }
            min = 999999;
            cenx[j] = ceny[j] = 0;
            // For further calculation as it would be overwritten
        }
        //Calculating the new centroid coordinates
        for(int m = 0; m < k; m++) { // cluster number
            xcnt=0;
            for(int j = 0; j < n; j++) { //data objects
                if(kc[j] == m) {
                    cenx[m] = x[j]+cenx[m];
                    ceny[m] = y[j]+ceny[m];
                    xcnt++;
                }
            }
            if(xcnt != 0){
               cenx[m] = cenx[m]/xcnt;
               ceny[m] = ceny[m]/xcnt;
            }
        }
        it++;
    }while(it < max_it);
}
```

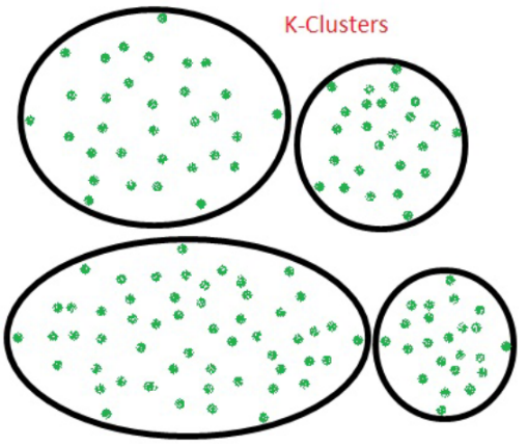


Fig. 1. K-Clusters

Now, we will find a convex hull for each cluster formed using the `findHull` function.

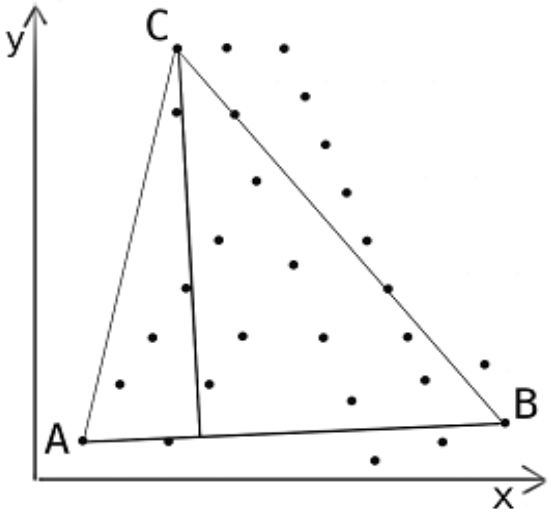
```
//for each cluster find the convex hull and store the points
int con_x[n],con_y[n],con_n=0,fin_x[n],fin_y[n],fin_n=0;

for(int j = 0; j < k; j++){
    findHull(xc[j], yc[j], clust_size[j], con_x, con_y, &con_n);
    printf("\n\nCluster %d; tid=%d\n", j+1, omp_get_thread_num());
    for(int i = 0; i < con_n; i++){
        fin_x[fin_n+i] = con_x[i];
        fin_y[fin_n+i] = con_y[i];
        printf("(%d,%d)\n",con_x[i],con_y[i]);
    }
    fin_n += con_n;
    printf("l_n=%d,g_n=%d\n",con_n,fin_n );
}
```

Here, `findHull()` calculates the minimum and maximum x-coordinates and passes them as arguments to call the `quickHull()`. `quickHull()` is used to find the Convex Hull incorporating the QuickHull algorithm.

Quick Hull Algorithm:

- Find the point with minimum x-coordinate, let's say, `min_x` and similarly the point with maximum x-coordinate, `max_x`.
- Make a line joining these two points, say L. This line will divide the whole set into two parts. Take both the parts one by one and proceed further.
- For a part, find the point P with maximum distance from the line L. P forms a triangle with the points `min_x`, `max_x`. It is clear that the points residing inside this triangle can never be the part of a convex hull.
- The above step divides the problem into two sub-problems (solved recursively). Now the line joining the points P and `min_x` and the line joining the points P and `max_x` are new lines and the points residing outside the triangle are the set of points. Repeat point no. 3 till there is no point left with the line. Add the end points of this point to the convex hull.



FINDHULL Function implementation

```
/**
 * @description Finds point with min and max x-coordinate, use them for as
arguments for quickHull function
 * @param x, y array of coordinates
 * @param [in] n number of points in a cluster
 * @param convx : x-coordinates of hull of a cluster
 * @param convx : y-coordinates of hull of a cluster
 * @return null
 */

void findHull(int x[], int y[], int n, int *convx, int *convy, int *n_c){
    int hullx[100000],hully[100000],index=0;
    // a[i].second -> y-coordinate of the ith point
    if (n < 3){
        printf("Convex hull not possible\n");
        return;
    }

    // Finding the point with minimum and maximum x-coordinate
```

```
    int min_x = 0, max_x = 0;
    for (int i = 1; i < n; i++) {
        if (x[i] < x[min_x]) min_x = i;
        if (x[i] > x[max_x]) max_x = i;
    }

    // Recursively find convex hull points on
    // one side of line joining a[min_x] and a[max_x]

    index = quickHull(x,y,n,x[min_x],y[min_x],x[max_x],y[max_x],
1,hullx,hully,index);

    // Recursively find convex hull points on
    // other side of line joining a[min_x] and a[max_x]

    index = quickHull(x,y, n, x[min_x],y[min_x], x[max_x],y[max_x],
-1,hullx,hully,index);

    int ind = setfunction(hullx, hully, index, convx, convy);

    *n_c = ind;
}
```

QUICKHULL Function Implementation

```
/**
 * @description
 * @param [in] n number of points in a cluster
 * @param convx : x-coordinates of hull of a cluster
 * @param convx : y-coordinates of hull of a cluster
 * @param { px, py } : start point coordinate of line
 * @param { qx, qy } : end point coordinate of line
 * @param side : can be 1 or -1 specifying each parts made by line L
 * @return index
 */

// End points of line L are p1 and p2.  side can have value
// 1 or -1 specifying each of the parts made by the line L
int quickHull(int x[],int y[], int n, int px, int py,int qx, int qy, int
```

```
side, int hullx[],int hully[],int index) {
    int ind = -1;
    int max_dist = 0;

    // finding the point with maximum distance
    // from L and also on the specified side of L.
    for (int i = 0; i < n; i++){
        int temp = lineDist(px,py, qx,qy, x[i],y[i]);
        if (findSide(px,py,qx,qy,x[i],y[i]) == side && temp > max_dist){
            ind = i;
            max_dist = temp;
        }
    }

    // If no point is found, add the end points of L to the convex hull.
    if (ind == -1){
        hullx[index] = px;
        hully[index] = py;
        index = index+1;
        hullx[index] = qx;
        hully[index] = qy;
        index = index+1;
        return index;
    }

    // Recur for the two parts divided by a[ind]

    index = quickHull(x,y,n, x[ind],y[ind],px,py,
-findSide(x[ind],y[ind],px,py, qx,qy),hullx,hully,index);
    index = quickHull(x,y, n, x[ind],y[ind],qx,qy,
-findSide(x[ind],y[ind], qx,qy, px,py),hullx,hully,index);

    return index;
}
```

```
int findSide(int p1x,int p1y,int p2x,int p2y,int px,int py){
    int val = (py - p1y) * (p2x - p1x) - (p2y - p1y) * (px - p1x);
    if (val > 0) return 1;
    if (val < 0) return -1;
    return 0;
}

// returns a value proportional to the distance between the point p and the
// line joining the points p1 and p2
int lineDist(int px,int py,int qx,int qy,int rx,int ry)
    return abs ((ry - py) *(qx - px) - (qy - py) * (rx - px));
```

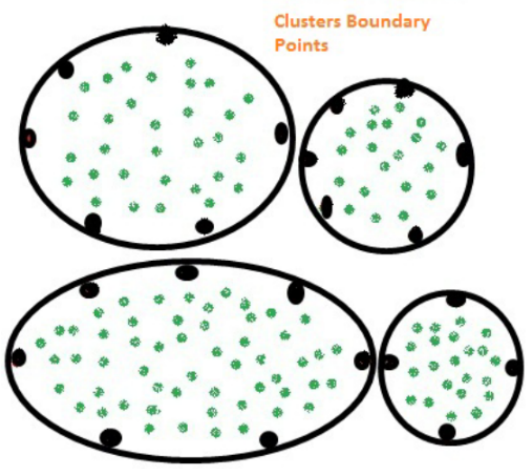


Fig. 2. Clusters Boundary Points

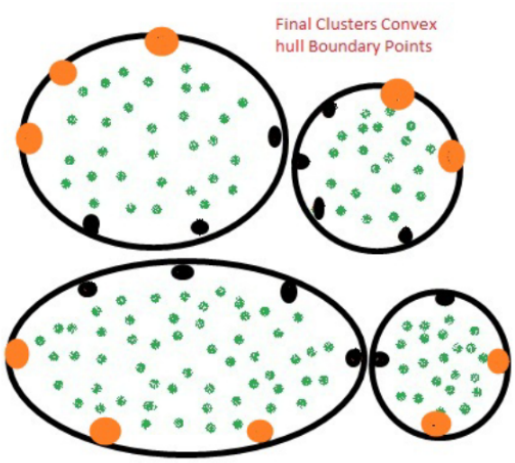


Fig. 3. Final Clusters Convex hull Boundary Points

Finding the final hull using all the final clusters convex hull boundary points. These boundary points will be considered as a new set of points to find the final hull using `findHull()`.

Calling `findHull()` Function for finding Final Hull

```
//now find the convex hull for the points stored above
printf("Finding the final Hull..!!\n");

for(int i=0;i<fin_n;i++) printf("(%d,%d)\n", fin_x[i], fin_y[i]);

printf("%d\n",fin_n);

findHull(fin_x, fin_y, fin_n, con_x, con_y, &con_n);

printf("***\n");

for(int i = 0; i < con_n; i++){
    printf("(%d,%d)\n",con_x[i],con_y[i]);
}
```

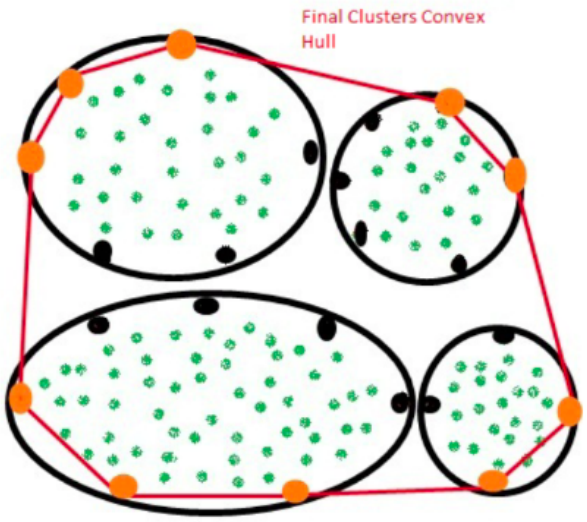


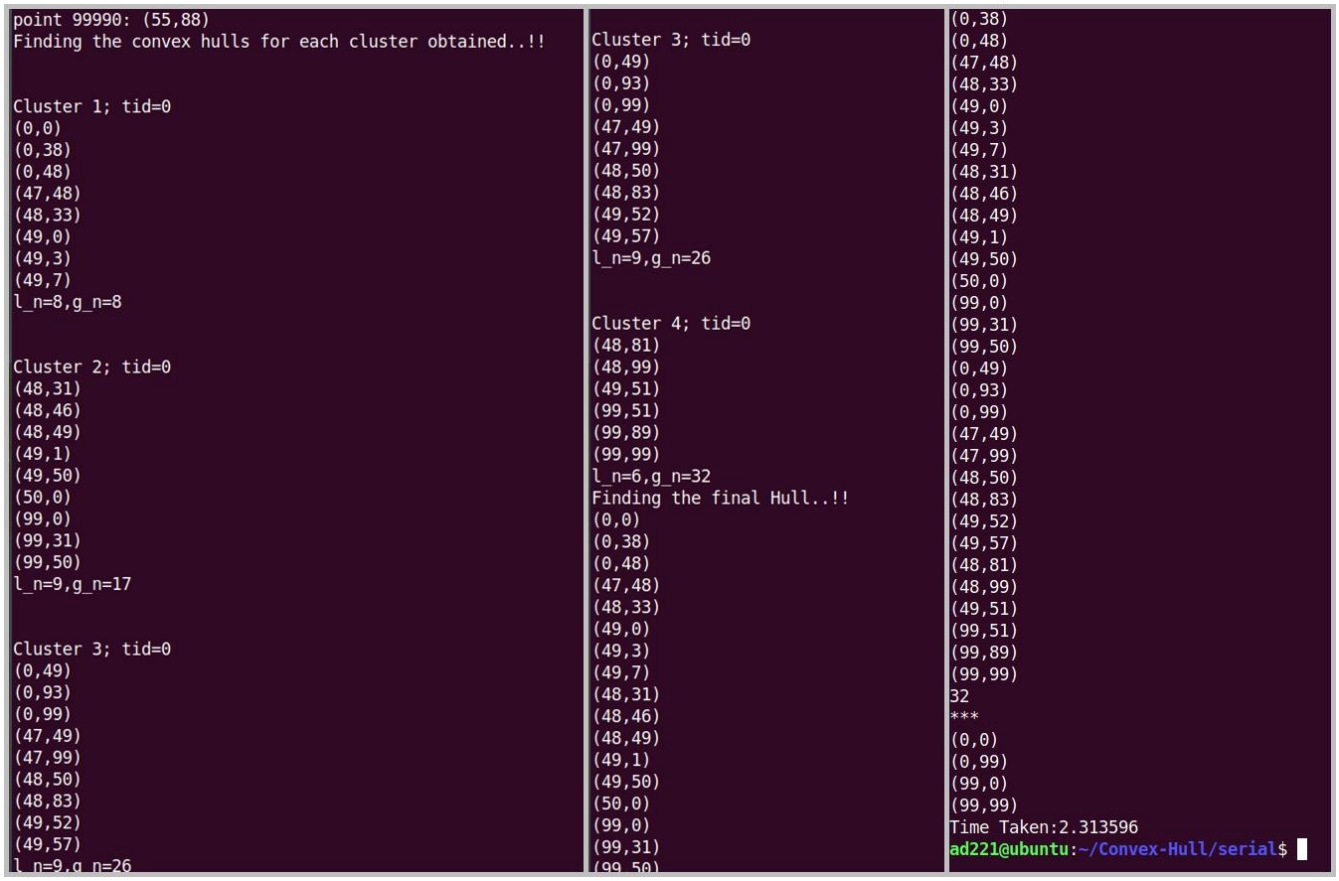
Fig. 4. Final Clusters Convex hull

Complexity

- Worst case time complexity: $O(n * n)$
- Average case time complexity: $O(n \log n)$
- Best case time complexity: $O(n \log n)$
- Space complexity:$O(n)$

Result

Input: n=100000 k=4 Avg Time: 2.29 seconds



-----X-----X-----X-----X----X----X-----X-----X-----X----X------