Parallel Implementation Solving Convex Hull Problem in 2D and 3D

Zesen Wang
zesen@kth.se

Ruiyang Ma
ruiyangm@kth.se

***Abstract*—The project aim at implementing state-of-art algorithms for solving 2D and 3D convex hull problem in a parallel method, and it performs experiments to examine the performance of the implementation in different data settings.**

I. Introduction

The definition of the convex hull problem in 2D is to find the smallest convex polygon that has all points inside it. The definition of the convex hull problem in 3D is to find the smallest convex polyhedron that has all points inside it.

In the 2D context, the well-known algorithms for solving convex hull include Graham's Scan [3], Chan's Algorithm [1], etc. And in the 3D context, the well-known algorithms are Divide and Conquer [7] and Incremental Convex Hull Algorithm [6]. All these sequential algorithms can obtain time complexity no more than $\mathcal{O}(N \log N)$.

When the scale of the problem is large, the run time and the memory of the sequential algorithms may be huge, which is not acceptable under some circumstances. Therefore, parallel algorithms to solve the convex hull problem play an important role in solving this problem.

In this project, our parallel implementation for the 2D convex hull problem mainly uses the idea of the parallel algorithm described in [5]. And for our parallel implementation solving the 3D convex hull problem, it mainly uses the idea of Incremental Convex Hull Algorithm [6] to solve the small cases, and it uses the idea of Divide and Conquer Algorithm [7] to combine the local results.

In the following sections, both sequential and parallel versions of algorithms will be introduced in detail.

II. Basic Concepts

A. Cross Product

The cross product $\boldsymbol{a} \times \boldsymbol{b}$ is defined as a vector $\boldsymbol{c}$ that is perpendicular (orthogonal) to both $\boldsymbol{a}$ and $\boldsymbol{b}$, with a direction given by the right-hand rule and a magnitude equal to the area of the parallelogram that the vectors span.

$$\boldsymbol{c} = \boldsymbol{a} \times \boldsymbol{b} = \|\boldsymbol{a}\|\|\boldsymbol{b}\| \sin\theta \cdot \boldsymbol{n}$$

where $\boldsymbol{n}$ is a unit vector perpendicular to the plane containing $\boldsymbol{a}$ and $\boldsymbol{b}$ in the direction given by the right-hand rule.

Even though the cross product is in 3D context, in 2D context, cross product can be used as a tool to check whether the directions of $\boldsymbol{a}$ and $\boldsymbol{b}$ rotate clockwisely or counterclockwisely by setting the both 3rd dimension values of $\boldsymbol{a}$ and $\boldsymbol{b}$ as 0 and checking the sign of the 3rd component of $\boldsymbol{c}$.

In 3D context, the magnitude of $\boldsymbol{c}$ divided by the product of the magnitudes of $\boldsymbol{a}$ and $\boldsymbol{b}$ can represent the sine value of the angle between $\boldsymbol{a}$ and $\boldsymbol{b}$.

B. Supporting Line

Supporting line is a concept defined in 3D context. When merging two 3D convex hulls, the supporting line includes lines that are on the combined convex hull and their two nodes of lines are on two original convex hulls respectively. The following figure shows one of supporting lines. It's important because it will be used as a starting edge to find the facets that connect two convex hulls, which will be introduced in the algorithm section.

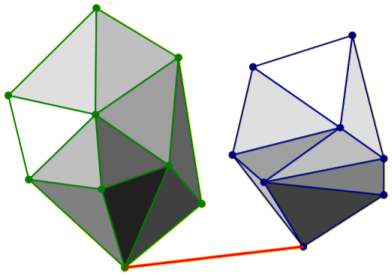

Fig. 1. Supporting Line [2]

III. Sequential Algorithms

A. 2D Convex Hull

1) Graham's Scan: The algorithm firstly finds the point with lowest y-coordination (denote as A) which is known to be on the convex hull. Then it sorts other points according to the angle between the vector $\overrightarrow{AX}$ (X is one of the other points) and positive x-axis.

Then it uses a stack structure to store the points on convex hull. The point A and the first point of sorted result are put in the stack as initialization. Then points are added to the stack sequentially. If the newly pushed point, with the top-2 points in the stack, does not satisfy the property of the convex hull, then pop the topmost point in the stack. Repeat this operation until all points are ever pushed to the stack.

The following pseudocode is adapted from Sedgewick and Wayne's Algorithms, 4th edition.

```
let N           = number of points
let points[N+1] = the array of points
swap points[1] with the point with the lowest y-coordinate
sort points by polar angle with points[1]

# We want points[0] to be a sentinel point that will
# stop the loop.
let points[0] = points[N]

# M will denote the number of points on the convex hull.
let M = 1
for i = 2 to N:
    # Find next valid point on convex hull.
    while ccw(points[M-1], points[M], points[i]) <= 0:
        if M > 1:
            M -= 1
            continue
        # All points are collinear
        else if i == N:
            break
        else
            i += 1
    # Update M and swap points[i] to the correct place.
    # This code is wrong as explained in the talk page.
    # When M and i are the same, the algorithm ends up
    # in an infinite loop.
    M += 1
    swap points[M] with points[i]
```

The time complexity for sorting points according to the angle is $\mathcal{O}(N \log N)$, and the time complexity for constructing the convex hull is $\mathcal{O}(N)$. Therefore, the total time complexity is $\mathcal{O}(N \log N)$.

B. 3D Convex Hull

1) Incremental Convex Hull Algorithm: The idea of the Incremental Convex Hull Algorithm is to construct a simple tetrahedron at first. Then the algorithm iterates on other nodes. When a node is on the scene, firstly judge whether the node is inside the current convex polyhedron. If a node is outside the convex hull, then delete the facets that are visible by the node and construct new facets between the node and the visible bound.

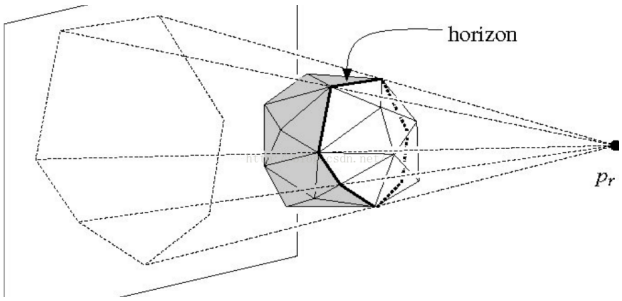

Fig. 2. Incremental Convex Hull Algorithm [8]

Figure 2 shows the main idea of the algorithm. When all nodes are processed, the final convex hull is constructed.

In detail, the process of deleting visible facets and the visible bound is implemented using flood-fill algorithm, which means when a facets is visible, then delete all adjacent facets that are visible because the visible facets are continuous.

The time complexity of the algorithm depends on the order of the points processed. A paper shows that the complexity of the Incremental Convex Hull Algorithm in 3D is $\mathcal{O}(N \log N)$ [4], but it may be unstable.

2) Divide & Conquer Algorithm: Firstly, the algorithm recursively divides the nodes in two parts until the number of nodes is sufficiently small. Then it constructs basic convex hull based on the small number of nodes. The main point of the idea of the Divide & Conquer Algorithm in 3D is on the merge part. The next step is to recursively merge the adjacent convex hulls.

The process of merging two 3D convex hulls is as follows.

1) Find one supporting line: The algorithm is to project the two 3D convex hulls to xy-plane respectively. Then construct the 2D convex hull on two 3D convex hulls' projections separately. The algorithm for constructing 2d convex hull takes $\mathcal{O}(M)$ (M is the number of nodes of one 3d convex hull) time complexity using the connection relationship in 3D because (1) if points are on 2D convex hull, then they must be on 3D convex hull (2) if two points are adjacent on the constructed 2D convex hull, then they must be adjacent on 3D convex hull.
   After the 2D convex hulls are constructed, find the leftmost point of the right convex hull and the rightmost point of the left convex hull. Then, as indicates in Figure 3, clockwisely and counter-clockwisely rotate two pivots respectively until one more operation will lead to a invalid connecting line.

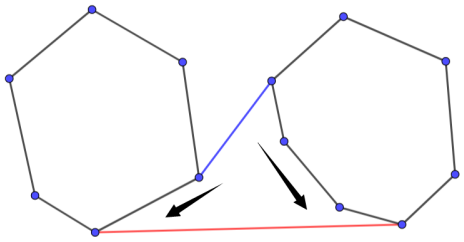

Fig. 3. Find a Supporting Line

   Finally, a supporting line is found.
2) Next step is to construct the facets that connect two 3D convex hull. It mainly uses the idea of gift wrapping, which means in every step, it finds the facet that has the smallest angle between the last one. It has to be mentioned that when in the first time to find the next facet, there is no last facet. Therefore, the facet that includes the supporting line and that is orthogonal to xy-plane is used as the last plane when in the first step.

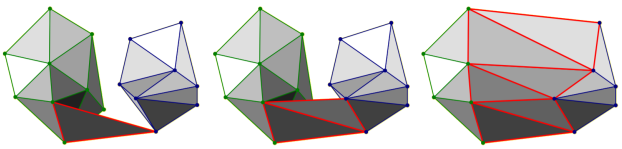

Fig. 4. Gift Wrapping [2]

3) The last step is to delete the facets that are not visible anymore. The method is flood-fill. In the last step, the bounds for visible and invisible facets are found. Therefore, a flood-fill is performed starting from any invisible facets, then all invisible facets get deleted.

When we are implementing this algorithm, we use Incremental Convex Hull Algorithm to deal with the small cases (with less than 200 points) whose time complexity can be treated as constant. And for the merge part, the time complexity is $O(M)$ where M is the number of points on the 2 convex hulls. Therefore, the total time complexity for the Divide & Conquer Algorithm is $\mathcal{O}(N \log N)$.

IV. 2D Convex Hull Algorithm in Parallel

A. Algorithm

The idea of algorithm is from [5].

The algorithm firstly splits all data points into P (the number of processors) parts (load-balanced linear data distribution) according to x-coordinate. Then every processor deals with their data locally using Graham's Scan (introduced in section III-A1). Then every processor communicates with each other to get the information of others' convex hulls.

Then the next step is to merge the convex hulls produced by each processor. Before introducing the merging part, some concepts and notations should be mentioned.

1) Upper/Lower hull: The convex hull is partitioned, using the leftmost and the rightmost points, into the upper hull and the lower hull as illustrated in Figure 5.

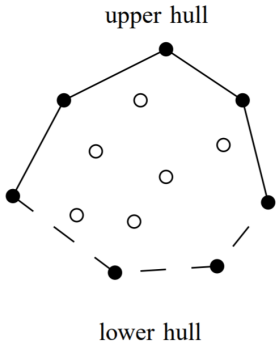

Fig. 5. Upper/Lower Hull [5]

2) $c_u(i,j)$ stands for the local index of the point on convex upper hull i that is one of the end of common tangent between i^{th} convex upper hull and j^{th} convex upper hull. And $c_l(i,j)$ has the similar definition as $c_u(i,j)$ while it is on the lower hull.

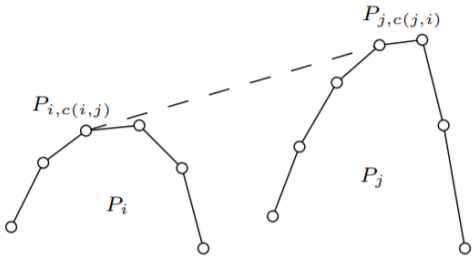

Fig. 6. Illustration of c_u [5]

For the merge part,

1) Calculate every $c_u(i,j)$ and $c_l(i,j)$: Take the case that $i < j$ as an example (same as the case in Figure 6). Firstly, a binary search is applied on i^{th} upper convex hull to find out the leftmost point that can have a line connecting to any point on j^{th} upper convex hull. This binary search works because of the property that as the position of the point is more left, the point is less likely to have a valid line connecting to j^{th} upper convex hull. The next step is to test whether the selected point on the i^{th} upper convex hull is able to have a line connecting to j^{th} upper convex hull. Then another binary search is applied on the j^{th} upper convex hull to find out the rightmost point that can connect to the point on i^{th} convex hull. If the found point on j^{th} upper convex hull and the selected point on i^{th} upper convex hull construct a common tangent, then the $c_u(i,j)$ and $c_u(j,i)$ are found.
The $c_l(i,j)$ and $c_l(j,i)$ can be found using the similar method.
2) Construct the combined convex hull: It's obvious that when $i < j$, the points on i^{th} upper convex hull with local index larger than $c(i,j)$ are not part of the final convex hull. Therefore, the local index interval of the points of i^{th} upper convex hull on the final convex hull is

$$l(i) = \max_{j<i}\{c(i,j)\}, r(i) = \min_{j>i}\{c(i,j)\}$$

$$\mathrm{I}_i = \begin{cases} \emptyset & (l(i) > r(i)) \\ [l(i), r(i)] & \text{otherwise} \end{cases}$$

3) The last step is to collect all $r(i)$ and $l(i)$ to generate the final convex hull.
It has to mention that according to the algorithm of [5], the algorithm may occur to error under the case in Figure 7. For the convex hull in middle, its $l(i)$ is equal to its $r(i)$, but point A is obvious not part of the combined convex hull. Therefore, a scan should be applied on the final convex hull to filter out the concave part.

B. Time Complexity Analysis

The time complexity for sorting points is $\mathcal{O}(N \log N)$. The time complexity to exchange all information about the local convex hulls is $\mathcal{O}(N)$. The time complexity to calculate $l(i)$

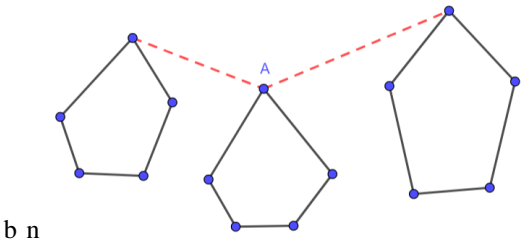

Fig. 7. Illustration of Possible Error

and $r(i)$ is $\mathcal{O}(N\log(\frac{N}{P})\log(\frac{N}{P}))$. The time complexity for generating the final convex hull is $\mathcal{O}(N)$.

Therefore, the total time complexity for the algorithm is $\mathcal{O}(N\log(\frac{N}{P})\log(\frac{N}{P}))$.

V. 3D Convex Hull Algorithm in Parallel

A. Algorithm

The algorithm is modified from [7]. The main process of the algorithm is

1) Split the points to P processors according to the x-coordinate. For every processor, it constructs 3d convex hull using the algorithm described in section III-B2.
2) The next step is to merge the convex hulls calculated in each processor.

```
let D = ceil(log P)
for i = 1 to D:
    let dst = flip(p, i)
    if dst < p:
        send local convex hull to processor dst
    if (dst > p) and (dst < P):
        receive local convex hull from processor dst
        merge the received one with the local one
if p = 0:
    return local convex hull
```

The "merge" step used here is as same as the merge part in section III-B2.

B. Complexity Analysis

The time complexity for local 3d convex hull construction is $\mathcal{O}(N\log N)$. The total time complexity for communication is $\mathcal{O}(N)$. The total time complexity for merging convex hulls from other processor is $\mathcal{O}(N)$. Therefore, the total time complexity for the parallel algorithm is $\mathcal{O}(N\log N)$.

VI. Experiment & Result

Since the communication time depends largely on the number of nodes on local convex hulls on each processor (output-sensitive algorithm), the programs are designed to run on several different circumstances.

A. 2 Dimension

The three different datasets are square, disk and circle, which are shown in Figure 8.

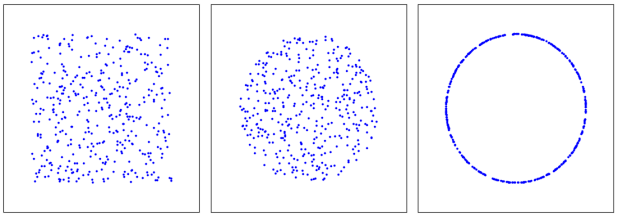

Fig. 8. Data Settings for 2D

P	Square (s)	S_P	Disk (s)	S_P	Circle (s)	S_P
1	1.6260e+00	-	1.6129e+00	-	8.5803e-01	-
2	8.2344e-01	1.97	8.5469e-01	1.89	4.2580e-01	2.02
3	5.5698e-01	2.92	5.6852e-01	2.84	2.9308e-01	2.92
4	4.1962e-01	3.87	4.2607e-01	3.79	2.2753e-01	3.77
5	3.4474e-01	4.72	3.5725e-01	4.51	1.8797e-01	4.56
6	2.8556e-01	5.69	3.0185e-01	5.34	1.6221e-01	5.29
7	2.5814e-01	6.30	2.6472e-01	6.09	1.4950e-01	5.74
8	2.2384e-01	7.26	2.2743e-01	7.09	1.2686e-01	6.76

TABLE I
Results for Runtime

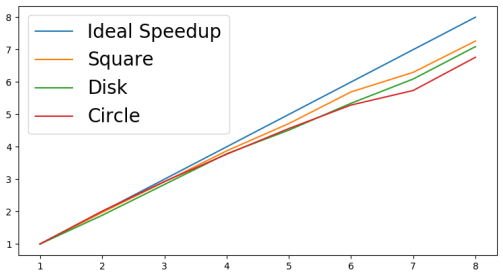

Fig. 9. Results for Speedup (2D)

We made experiments on different datasets with $N = 10^7$. The runtimes and speedups for different number of processors are shown in Table I and Figure 9.

The result shows acceptable speedups for all cases. And as the number of points on local convex hulls becomes larger, the speedup starts to getting small. Also, it shows that the speedup becomes smaller as the number of processors increases.

The result of experiment shows exactly what we expected.

B. 3 Dimension

The three different datasets are cube, ball and ellipsoid, which are shown in Figure 10.

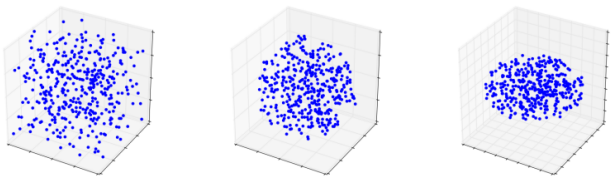

Fig. 10. Data Settings for 3D

The experimental setting for N is 10^6. The runtimes and speedups for different number of processors are shown in Table II and Figure 11.

P	Cube (s)	S_P	Ball (s)	S_P	Ellipsoid (s)	S_P
1	1.9630e+00	-	2.1031e+00	-	2.1262e-00	-
2	1.0004e-00	1.96	1.0819e-00	1.94	1.0973e-00	1.93
3	6.9005e-01	2.84	7.4564e-01	2.82	7.6094e-01	2.79
4	5.0746e-01	3.86	5.4200e-01	3.88	5.5132e-01	3.85
5	4.5366e-01	4.33	4.9227e-01	4.27	5.0586e-01	4.20
6	3.7057e-01	5.30	4.0094e-01	5.25	4.1403e-01	5.14
7	3.1589e-01	6.21	3.4804e-01	6.04	3.4582e-01	6.15
8	2.8129e-01	6.98	3.0349e-01	6.93	3.1669e-01	6.71

TABLE II
RESULTS FOR RUNTIME

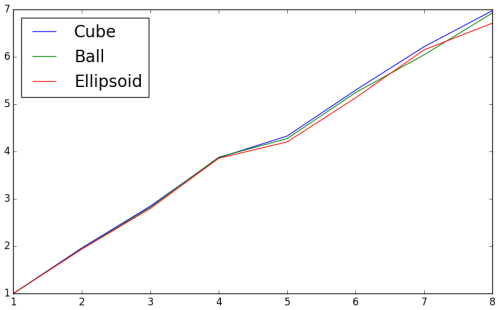

Fig. 11. Results for Speedup (3D)

The result shows acceptable speedups for all cases. And as the number of points on local 3D convex hulls becomes larger, the speedup starts to getting small. Also, it shows that the speedup becomes smaller as the number of processors increases.

The result of experiment shows exactly what we expected.

VII. CONCLUSION

The project implements sequential and parallel version of convex hull algorithm in 2D context and 3D context.

The results show that the implementation obtain acceptable speedup. Also, multiple datasets are applied on the algorithm to test the performance of the algorithm under different circumstances. Our implementation gets correctly verified. Generally speaking, the implementation and the experiments are successful.

In this project, we learn some advanced idea to solve 2D and 3D convex hull problem and to solve the problem in parallel version.

REFERENCES

[1] Chan, Timothy M. "Optimal output-sensitive convex hull algorithms in two and three dimensions." *Discrete & Computational Geometry* 16.4 (1996): 361-368.

[2] "Convex Hulls in 2D and 3D." *2-3D-enveloppe-convexe-od*, Loria, https://members.loria.fr/MPouget/files/enseignement/webimpa/2-3D-enveloppe-convexe-od.pdf. Accessed 16 Apr. 2018.

[3] Graham, Ronald L. "An efficient algorith for determining the convex hull of a finite planar set." *Information processing letters* 1.4 (1972): 132-133.

[4] Kallay, Michael. "The complexity of incremental convex hull algorithms in Rd." *Information Processing Letters* 19.4 (1984): 197.

[5] Nakagawa, Masaya, et al. "A simple parallel convex hulls algorithm for sorted points and the performance evaluation on the multicore processors." *Parallel and Distributed Computing, Applications and Technologies, 2009 International Conference on*. IEEE, 2009.

[6] o'Rourke, Joseph. Computational geometry in C. *Cambridge university press*, 1998.

[7] Preparata, Franco P., and Se June Hong. "Convex hulls of finite sets of points in two and three dimensions." *Communications of the ACM* 20.2 (1977): 87-93.

[8] *3D Incremental Convex Hull Algorithm*, 9 Feb. 2016, https://blog.csdn.net/theArcticOcean/article/details/50646428. Accessed 16 Apr. 2018.