CONVEX HULL - PARALLEL AND DISTRIBUTED ALGORITHMS

Jayanth Ramesh, Suhas Suresha
{jayanth7, suhas17}@stanford.edu

***Abstract*—In this project, we examine two different algorithms for finding the convex hull. We analyze their performance in the sequential world and discuss about how well they can be parallelized and how well they scale in the distributed scenario.**

I. Introduction

CONVEX hull of a set of points S is the smallest convex set that contains S. A convex hull is also known as convex envelope. It is a very interesting problem that has applications in a wide variety of fields ranging from image processing to game theory [4]. This wide ranging real world application motivated us to study the algorithms for computing convex hulls of a set of planar points and understand their performance.

In this project, we consider two popular algorithms for computing convex hull of a planar set of points. The first algorithm is The Ultimate Planar Convex Hull Algorithm, which was proposed by David G. Kirkpatrick and Raimund Seidel [2] [1]. The second algorithm is the Quick Hull algorithm [3] which was discovered independently in 1977 by W. Eddy and in 1978 by A. Bykat [5]. We discuss the amenability of these algorithms to the parallel and distributed scenario. We consider the PRAM model for the parallel scenario.

The rest of the paper is structured as follows. We first discuss the sequential version of The Ultimate Planar Convex hull algorithm in detail, proving correctness and analyzing its complexity, followed by a discussion about the performance of the algorithm in the parallel and distributed world. We then present the implementation of QuickHull algorithm, examining its performance in the sequential, parallel and distributed scenarios.

II. Ultimate Planar Convex Hull Algorithm

In this section, we discuss the sequential and parallel implementation of the Ultimate Planar Convex Hull Algorithm in detail. We also prove correctness and analyze the algorithm complexity for both sequential and parallel versions of the algorithm.

Ultimate Planar Convex Hull Algorithm employs a divide and conquer approach. It computes the upper convex hull and lower convex hull separately and concatenates them to find the Convex Hull. The overview of the algorithm is given in Planar-Hull(S).

Since an algorithm for constructing the upper convex hull can be easily modified to obtain the lower convex hull, we only discuss the Upper-Hull algorithm here. The Upper-Hull(S), given below, returns the sequence of vertices on the upper convex hull. First, the algorithm finds a vertical line that divides the given point set S into two approximately equal parts. It then determines the edge of the Upper Hull crossing this vertical line, called as the "bridge". It then eliminates the points that lie underneath the bridge, and then applies the algorithm recursively to the two sets of remaining points on the left and right hand side of the vertical line to find the upper convex hull.

Algorithm 1 Planar-Hull(S)

```
1: #S is a dictionary mapping node id to $(x, y)$ co-ordinates
2: **return** Upper-Hull(S) + Lower-Hull(S)
```

Algorithm 2 Upper-Hull(S)

```
1: #S is a dictionary mapping node id to $(x, y)$ co-ordinates
2: Initialization: Determine $p_{min}$ and $p_{max}$ where $x(p_{min}) \leq x(p_i) \leq x(p_{max})$. If $x(p_i) = x(p_{min})$, then $y(p_{min}) \geq y(p_i)$. If $x(p_i) = x(p_{max})$, then $y(p_{max}) \geq y(p_i)$.
3: **if** $p_{min} = p_{max}$ **then**
4:     **print** $p_{min}$ and stop
5: **end if**
6: Let $T = \{p_{min}, p_{max}\} \bigcup \{p \in S | x(p_{min}) < x(p_i) < x(p_{max})\}$
7: CONNECT($p_{min}$,$p_{max}$,T)
```

The most complex part of this algorithm is to find the bridge crossing the vertical line. The function BRIDGE takes as parameters a set of points and a real number a representing the vertical line $L = \{(x, y)|x = a\}$. It returns as output a pair (p_i, p_j), where p_i and p_i are the left and right bridge point respectively. The algorithm uses the properties of convex hull to recursively eliminate points that are not bridge points.

In the following subsections, we prove the correctness of the Planar Hull algorithm and discuss the complexity for both sequential and parallel implementation.

A. Correctness of the Algorithm

It is obvious that the Planar-Hull algorithm correctly returns the convex hull of the given set of points. Now, we will prove the correctness of the Upper-Hull algorithm assuming that the BRIDGE method gives the desired output. We will then prove the correctness of the BRIDGE method.

Algorithm 3 CONNECT(k, m, S)

```
#S is a dictionary mapping node id to $(x, y)$ co-ordinates
Find real number $a$ such that $x(p_i) \leq a$ for $\lceil |S|/2 \rceil$ points in $S$ and $x(p_i) \geq a$ for $\lceil |S|/2 \rceil$ points in $S$.
$(p_i, p_j)$ = BRIDGE($S, a$)
Let $S_{left} = \{p_i\} \bigcup \{p \in S | x(p) < x(p_i)\}$
Let $S_{right} = \{p_j\} \bigcup \{p \in S | x(p) > x(p_j)\}$
if $p_i = p_k$ then
    print $p_i$
else
    CONNECT(k,$p_i$,$S_{left}$)
end if
if $p_j = p_m$ then
    print $p_j$
else
    CONNECT(j,$p_m$,$S_{right}$)
end if
```

If the upper hull of the set of points S consists of only one vertex, i.e, if all of S lies on one vertical line, then the algorithm Upper-Hull is trivially correct and reports that vertex. Otherwise, it calls the method CONNECT which finds the bridge (p_i, p_j), which is an edge on the upper convex hull. If p_i turns out to be the leftmost vertex of the upper hull, it will be printed. Otherwise, the recursive call CONNECT will cause the sequence of vertices from p_{min} to p_i on the upper hull to be printed. Similarly, if p_j turns out to be the rightmost vertex of the upper hull, it will be printed. Otherwise, the recursive call CONNECT will cause the sequence of vertices from p_j to p_{max} on the upper hull to be printed.

Now in order to prove the correctness of the BRIDGE method, we need to state a few lemmas. First we define a supporting line of S to be a non-vertical straight line which contains at least one point of S but has no points of S above it. It is obvious that the bridge must be contained in some supporting line. Let's call that particular line as b and let s_b be the slope of b. In the BRIDGE method, we pair up the points of S into $\lfloor |S|/2 \rfloor$ couples. The following lemmas show how forming pairs of points can help us eliminate candidates as bridge points.

Lemma 2.1: Let p, q be a pair of points of S. If $x(p) = x(q)$ and $y(p) > y(q)$, then q cannot be a bridge point.

Proof: Since the bridge is an edge of the upper convex hull, q cannot be a bridge point as it cannot be a part of the upper convex hull. ∎

Lemma 2.2: Let p, q be a pair of points of S with $x(p) < x(q)$, and let s_{pq} be the slope of the straight line h through p and q.

(1) If $s_{pq} > s_b$ then p cannot be a bridge point.

(2) If $s_{pq} < s_b$ then q cannot be a bridge point.

Proof: Assume that $s_{pq} > s_b$ is true and p is a bridge point. Since $s_{pq} > s_b$ and $x(p) < x(q)$, q will lie above the bridge line b which would contradict the fact the b is a supporting line of S. Therefore p cannot be a bridge point. A similar proof follows for case (2) as well. ∎

Algorithm 4 BRIDGE(S, a)

```
#S is a dictionary mapping node id to $(x, y)$ co-ordinates
$CANDIDATES = \emptyset$
if $|S| = 2$ then
    return $((p_i, p_j))$, where $x(p_i) < x(p_j)$
end if
Choose $\lfloor |S|/2 \rfloor$ disjoint sets of size 2 from S. If a point of S remains, add it to $CANDIDATES$.
Arrange each subset to be an ordered pair $((p_i, p_j))$ such that $x(p_i) \leq x(p_j)$. Name this set of ordered pairs as $PAIRS$.
for all $(p_i, p_j)$ in $PAIRS$ do
    if $x(p_i) = x(p_j)$ then
        Delete $(p_i, p_j)$ from $PAIRS$
        if $y(p_i) > y(p_j)$ then
            Insert $p_i$ into $CANDIDATES$
        else
            Insert $p_j$ into $CANDIDATES$
        end if
    else
        Let $k(p_i, p_j) = (y(p_i) - y(p_j))/(x(p_i) - x(p_j))$
    end if
end for
if $PAIRS = \emptyset$ then
    return BRIDGE($CANDIDATES, a$)
end if
Determine K, the median of $\{k(p_i, p_j) | (p_i, p_j) \in PAIRS\}$
Let $SMALL = \{(p_i, p_j) \in PAIRS | k(p_i, p_j) < K\}$
Let $EQUAL = \{(p_i, p_j) \in PAIRS | k(p_i, p_j) = K\}$
Let $LARGE = \{(p_i, p_j) \in PAIRS | k(p_i, p_j) > K\}$
Let $MAX$ be the set of points $p_i \in S$ such that $y(p_i) - K * x(p_i)$ is maximum.
Let $p_k$ be the point in $MAX$ with with minimum x-coordinate
Let $p_m$ be the point in $MAX$ with with maximum x-coordinate
if $x(p_k) \leq a$ & $x(p_m) > a$ then
    return $((p_k, p_m))$
end if
if $x(p_m) \leq a$ then
    for all $(p_i, p_j) \in LARGE \bigcup EQUAL$ do
        Insert $p_j$ into $CANDIDATES$
    end for
    for all $(p_i, p_j) \in SMALL$ do
        Insert $p_i$ and $p_j$ into $CANDIDATES$
    end for
end if
if $x(p_k) > a$ then
    for all $(p_i, p_j) \in SMALL \bigcup EQUAL$ do
        Insert $p_i$ into $CANDIDATES$
    end for
    for all $(p_i, p_j) \in LARGE$ do
        Insert $p_i$ and $p_j$ into $CANDIDATES$
    end for
end if
return BRIDGE($CANDIDATES, a$)
```

These two lemmas are really useful to eliminate at least one point from every one of the $\lfloor |S|/2 \rfloor$ couples. However it is hard to test the condition $s_{pq} > s_b$ without knowing the value of s_b, the slope of bridge line b, which is what we want to compute. The following lemma (has a trivial proof) suggests a simple solution to get around this difficulty.

Lemma 2.3: Let h be a supporting line of S with slope s_h.

(1) $s_h < S_b$ *iff* h only contains points of S that are strictly to the right of the vertical line L.
(2) $s_h = S_b$ *iff* h only contains a point of S that is strictly to the right of L and a point of S that is strictly to the left of L. This implies that h is the bridge line b.
(3) $s_h > S_b$ *iff* h only contains points of S that are strictly to the left of the vertical line L.

In the algorithm, we choose s_h to be K, which is the median of the slopes defined by the $\lfloor |S|/2 \rfloor$ pairs of points. Using lemma 2.3, we check if our supporting line h is the bridge. If not, we check if the points on the supporting line h lie on the right hand side or left hand side of the vertical line. If they lie on the right hand side, then for all pairs (p_i, p_j) having slope less than or equal to K, p_j cannot be a bridge point. If they lie on the left hand side, then for all pairs (p_i, p_j) having slope greater than or equal to K, p_i cannot be a bridge point. Hence the BRIDGE algorithm recursively gets rid of points that it confirms as not being the candidate points to return the bridge.

B. *Complexity Analysis for Sequential Version*

We first discuss the sequential complexity of the BRIDGE method. Operations from steps 3 to 19 in the BRIDGE algorithm, where the method creates $\lfloor |S|/2 \rfloor$ disjoint pairs of points and calculates their slopes, is executed in $\mathcal{O}(n)$ time. We can determine the median K in step 23 in $\mathcal{O}(n)$ time from Blum's algorithm. Steps 24 to 29 is executed in$\mathcal{O}(n)$ time. BRIDGE either finds the bridge points in step 31 or discards redundant points of S from steps 34 to 48 to recursively call BRIDGE again. Now since K is the median of the slopes of all the $\lfloor |S|/2 \rfloor$ disjoint pairs, there are $\lfloor |S|/4 \rfloor$ pairs having slope greater than or equal to K and $\lfloor |S|/4 \rfloor$ pairs having slope less than or equal to K. Depending on the side of L that the supporting line has its points, the method either executes steps 33 to 40 or steps 41 to 48. In either case, we eliminate one point from at least $\lfloor |S|/4 \rfloor$ pairs. Hence we pass at most $(3/4) * |S|$ points to the next recursive call. Hence the recurrence relation is

$$f(n) = f(3n/4) + \mathcal{O}(n) \tag{1}$$

It can be easily seen that such a recursive function has complexity $\mathcal{O}(n)$. Hence BRIDGE method has sequential time complexity of $\mathcal{O}(n)$.

We now look at the sequential time complexity of Upper-Hull method. Finding p_{min} and p_{max} in step 2 of Upper-Hull method takes $\mathcal{O}(n)$ time. If upper hull consists of only one vertex, the algorithm trivially stops at step 4, otherwise it calls the CONNECT method. In the CONNECT method, it takes $\mathcal{O}(n)$ time to determine a and the bridge points (p_i, p_j) as we saw in the previous analysis. It then recursively calls CONNECT twice, one for the points on the left and one for points on the right. Since a is chosen such a way that it divides the point set S into 2 equal parts, the recursive calls of CONNECT will not have more than $\lceil |S|/2 \rceil$ points as input. In each recursive call, CONNECT discovers at least one unknown edge which is part of the upper convex hull. Assume that there are h edges in the upper convex hull. Then the recurrence relation is given by

$$f(n,h) = cn + \max_{h_l + h_r = h} \left(f(\frac{n}{2}, h_l) + f(\frac{n}{2}, h_r) \right) \tag{2}$$

With some analysis, we can see that such a recursive function has time complexity $\mathcal{O}(n \log h)$. Hence Upper-Hull method has sequential time complexity of $\mathcal{O}(n \log h)$.

Planar-Hull method involves concatenating the results from Upper-Hull and Lower-Hull to obtain the convex hull, which takes constant time. Since Lower-Hull is a simple modification to the Upper-Hull algorithm, its time complexity is $\mathcal{O}(n \log h)$ as well. Hence the Planar-Hull algorithm has a sequential time complexity of $\mathcal{O}(n \log h)$.

C. *Complexity Analysis for Parallel Version*

In this section, we discuss the depth complexity of the Planar-Hull algorithm. We first look at the BRIDGE method. Operations from 3 to 29 in BRIDGE, which involves finding the median, can be executed in $\mathcal{O}(\log^2 n)$ depth. As discussed before, we pass at most $(3/4) * |S|$ points to the next recursive call. Hence the recursive relation is

$$f(n) = f(3n/4) + \mathcal{O}(\log^2 n) \tag{3}$$

It can be easily seen that such a recursive function has complexity $\mathcal{O}(\log^3 n)$. Hence BRIDGE method has depth complexity of $\mathcal{O}(\log^3 n)$.

We now look at the depth complexity of Upper-Hull method. Finding p_{max} and p_{max} in step 2 of Upper-Hull method takes $\mathcal{O}(\log n)$ depth. In the CONNECT method, it takes $\mathcal{O}(\log^2 n)$ time to determine a and $\mathcal{O}(\log^3 n)$ to determine the bridge points (p_i, p_j) as we saw in the previous analysis. The 2 recursive calls of CONNECT can be run in parallel. Hence the recurrence relation is given by

$$f(n,h) = clog^3 n + \max_{h_l + h_r = h} \left(f(\frac{n}{2}, h_l), f(\frac{n}{2}, h_r) \right) \tag{4}$$

With some analysis, we can see that such a recursive function has worst-case time complexity $\mathcal{O}(\log^4 n)$. Hence Upper-Hull method has depth complexity of $\mathcal{O}(\log^4 n)$.

Planar-Hull method involves concatenating the results from Upper-Hull and Lower-Hull to obtain the convex hull, which takes constant time. Since Lower-Hull is a simple modification to the Upper-Hull algorithm, its depth complexity is $\mathcal{O}(\log^4 n)$ as well. Hence the Planar-Hull algorithm has a depth complexity of $\mathcal{O}(\log^4 n)$.

Using Brent's theorem, we can bound the time taken when p processors are available as $\frac{\mathcal{O}(n \log h)}{p} \leq T_p \leq \frac{\mathcal{O}(n \log h)}{p} + \mathcal{O}(\log^4 n)$.

D. Planar Convex Hull - Distributed

Being a divide and conquer algorithm, Planar-Hull is not amenable to distributed scenario. In general, divide and conquer algorithms are not suitable for distributed scenario due to very high communication costs. The Planar Convex Hull algorithm involves a call to a recursive function CONNECT, which itself invokes BRIDGE, another recursive function. This makes The Ultimate Planar Convex Hull a very recursive algorithm, which would involve a lot of communication between the machines, making its use in practice inviable.

III. QuickHull Algorithm

This algorithm is in principle is similar to the QuickSort algorithm. In each iteration, the algorithm picks the points that would belong to the convex hull (To give its analogy to QuickSort, in each call to QuickSort, the correct position of the pivot is determined) and eliminates those points that do not belong to the convex hull. This is repeated recursively (and then the array elements to the left and right of pivot are recursively sorted) till there are no more points to be considered. The algorithm is recursive in nature. The pseudocode of the algorithm is given below.

Algorithm 5 QuickHull(S)

```
1: # S is a dictionary mapping node id to $(x, y)$ co-ordinates
2: # Result is a global list containing the points in the convex hull
3: Compute $x_{min}$ and $x_{max}$. If multiple points have the same $x_{min}$ or $x_{max}$, pick any one.
4: Let $p_{min}$ and $p_{max}$ correspond to points with x-coordinate as $x_{min}$ and $x_{max}$ respectively.
5: Add all points with x-coordinate as $x_{min}$ or $x_{max}$ to Results and remove these points from S
6: QuickHull_helper(S, $p_{min}$, $p_{max}$)
```

Below, we describe the algorithm used to check if a point lies in a triangle

A. Correctness of the Algorithm

One of the key observations is the following lemma.

Lemma 3.1: Any point which is the farthest away from a line (on either side) joining any two points belonging to the convex hull will also belong to the convex hull.

Proof: This can be proved by contradiction. Consider the case, where the farthest point (call it p) from the line is not included in the convex hull. Let the axes be re-oriented such that the x-axis is aligned along the line joining the two points in the convex hull. Then, the point farthest away from this line will have the maximum or minimum y-coordinate among all the set of points. Now, since all the other points have the absolute value of their y-coordinate less than the absolute value of y-coordinate of p, no convex combination of points already in the hull can result in p. But this contradicts the definition of a convex hull. Hence, p also belongs to the convex hull. ■

From the above observation, we can infer that the points which are farthest away from the line (on either side) joining

Algorithm 6 QuickHull_helper(S, p_1, p_2)

```
1: Let $p_u$ and $p_b$ be the points which are farthest from the line joining $p_{min}$ and $p_{max}$, on either side of the line respectively.
2: if $p_u$ exists then
3:     Add $p_u$ to Results and remove it from S
4:     Remove all points that lie in a triangle formed by $p_1$, $p_2$, $p_u$
5: end if
6: if $p_b$ exists then
7:     Add $p_b$ to Results and remove it from S
8:     Remove all points that lie inside the triangle formed by $p_1$, $p_2$, $p_b$
9: end if
10: if S is empty then
11:     return
12: end if
13: if $p_u$ exists then
14:     QuickHull_helper(S, $p_u$, $p_1$)
15:     QuickHull_helper(S, $p_u$, $p_2$)
16: end if
17: if $p_b$ exists then
18:     QuickHull_helper(S, $p_b$, $p_1$)
19:     QuickHull_helper(S, $p_b$, $p_2$)
20: end if
```

Algorithm 7 CheckIfInTriangle(S, p_1, p_2, p)

```
1: $v_0 = p$
2: $v_1 = p_1 - p$
3: $v_2 = p_2 - p$
4: for Points in S do
5:     v - the current point under consideration
6:     $a = (det(v, v2) - det(v0, v2))/(det(v1, v2))$
7:     $b = -(det(v, v1) - det(v0, v1))/(det(v1, v2))$
8:     if
9:         thenif $(a > 0 and b > 0 and a + b < 1)$:
10:        Remove v from S
11:    end if
12: end for
13: for points in S do
14: end for
```

the points whose x-coordinates are the minimum and maximum respectively will belong to the convex hull. Also, the points with the minimum and maximum x-coordinate will belong to the convex hull.

Any point which lies inside the triangle formed by the points at the extreme (min and max x-coordinate) and the point farthest away from the line joining these two will not belong to the convex hull, because if that happens, then the points lying on or outside the triangle would not lie inside the convex hull.

Hence, at every step, the algorithm identifies at least one point belonging to the convex hull if it exists, and eliminates at least one point belonging to the convex hull if there is any such point. Therefore, when all the points are considered, the algorithm correctly identifies the set of points that belong to the convex hull.

B. Complexity Analysis for Sequential Version

Let h be the number of edges in the convex hull, which is also the number of vertices in the convex hull. Let n be the total number of points under consideration.

The steps from 3 to 6 in QuickHull(S) can be performed in linear time. The steps from 1 to 10 in QuickHull_helper() can be done in linear time. Now, since at least one point belonging to the convex hull is identified in each call to QuickHull_helper(), at most h calls to the functions would be made.

Hence, on the whole, the algorithm has a time complexity of $\mathcal{O}(nh)$

C. Complexity Analysis for Parallel Version

The QuickHull algorithm is inherently sequential, as each of the recursive calls made are not independent of one another as S gets updated each time.

The analysis of time complexity follows a similar recipe as the sequential one, except for the fact that the operations from 3 to 6 in QuickHull(S) and 1 to 10 in QuickHull_helper() can be done in $\mathcal{O}(logn)$ time. Again, at most h calls need to be made to this helper function, making the overall complexity to be $\mathcal{O}(hlogn)$, assuming infinite processors are available.

Using Brent's theorem, we can bound the time taken when p processors are available as $\frac{\mathcal{O}(nh)}{p} \leq T_p \leq \frac{\mathcal{O}(nh)}{p} + \mathcal{O}(hlogn)$

IV. Distributed QuickHull

The QuickHull algorithm is a good candidate when considering the distributed version, because the communication costs involved are greatly reduced. Let us assume there are m machines and that the data is distributed evenly, so each machine stores n/m of the points.

Algorithm 8 Distributed QuickHull (S)

```
1: All the individual machines compute the minimum and
   maximum x-coordinate among their dataset and send the
   result to the driver machine, which then computes the
   global minimum and maximum and broadcasts it to all
   the machines and adds it to the convex hull.
2: Once each machine has the information about these two
   points, they can compute the point farthest from the
   line joining these two points as described previously and
   send their respective results to the driver, which can then
   broadcast the global point that is the farthest from the line
   to all the machines and adds it to the convex hull.
3: Once each machine has all the vertices of the triangle,
   they can remove the points among their dataset that lie
   inside the triangle.
4: The recursive procedure can then be repeated in a similar
   fashion as described above.
```

A. Communication Cost

We observe that in the above algorithm, communication occurs only when we need to compute global maximums or minimums.

In each call to the QuickHull_helper(S, p_1, p_2), two computations of global maximums occur, which involves computing the point farthest from line joining p_1 and p_2 on either side of it. Therefore, each call to the helper function involves $\mathcal{O}(m)$ communications. This implies that the total communication cost is $\mathcal{O}(mh)$.

One important thing to observe here is that the communication cost scales only with the number of edges in the convex hull, and not with the number of points under consideration, which is a big win, because the number of edges in a convex hull are much less than the total number of points in general.

B. Time Complexity

1) Communication Time: Let L be the latency and B be the bandwidth. The overall communication cost for each round of computation of the global maximums of minimums would be $\mathcal{O}(L + \frac{m}{B})$. Since there would be h such rounds of all-reduce communication, the total communication time is $\mathcal{O}(hL + \frac{hm}{B})$

2) Work and Depth: The number of nodes stored in each machine is $\frac{n}{m}$, and each call to the QuickHull_helper() function involves $\mathcal{O}(number\ of\ points)$ work before making the recursive calls. Since all the machines, can do the work in parallel, we have the work done by each machine in each call to the helper function as $\mathcal{O}(\frac{n}{m})$. Since, there would be at most h such rounds of communication, the total work is given by $\mathcal{O}(\frac{n}{m}h)$.

The analysis of depth follows a similar recipe, as there isn't much scope for parallelization for the QuickHull algorithm. Each call to QuickHull_helper() would result in $\mathcal{O}(log(\frac{n}{m}))$ work before the recursive calls are made and hence the total depth is $\mathcal{O}(log(\frac{n}{m})h)$.

Using Brent's theorem, we can then bound the time it takes with p processors in each of the m machines as $\frac{\mathcal{O}(\frac{n}{m}h)}{p} \leq T_p \leq \frac{\mathcal{O}(\frac{n}{m}h)}{p} + \mathcal{O}(log(\frac{n}{m})h)$

V. Conclusion

We considered two algorithms for finding the convex hull and analyzed their time complexity for the sequential version of the algorithm. While the Ultimate Planar Convex Hull algorithm is a good candidate for a parallel algorithm, as there is a lot of scope for parallelization, it isn't amenable to the distributed scenario. The QuickHull algorithm on the other hand, while not a very good sequential algorithm (in terms of time complexity) or a parallelizable algorithm, is a good candidate for the distributed scenario, as the communication cost scale only with the number of edges in the convex hull and not with the number of points under consideration.

Algorithm	Sequential	Parallel	Distributed Comm. Cost
Ultimate Planar	$\mathcal{O}(nlogh)$	$\mathcal{O}(log^4 n)$	Inviable
QuickHull	$\mathcal{O}(nh)$	$\mathcal{O}(hlogn)$	$\mathcal{O}(mh)$

VI. Future Work

We showed two algorithms for finding convex hull, one which is good in the parallel scenario and another algorithm that is good in the distributed scenario. Future work could be aimed at coming up with an algorithm that utilizes the best of both these algorithms, which would then be a good candidate for both the parallel and distributed scenarios, along with good sequential time complexity.

References

[1] Akl, S. G. ”Optimal parallel algorithms for computing convex hulls and for sorting.” Computing 33.1 (1984): 1-11.
[2] Kirkpatrick, David G., and Raimund Seidel. ”The ultimate planar convex hull algorithm?.” SIAM journal on computing 15.1 (1986): 287-299.
[3] https://en.wikipedia.org/wiki/Quickhull.
[4] https://en.wikipedia.org/wiki/Convex_hull
[5] https://en.wikipedia.org/wiki/Convex_hull_algorithms