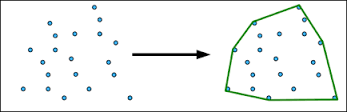
**Convex hull in 2D plane**

Akshay Miterani - 201401443

Mihir Limbachia - 201401456

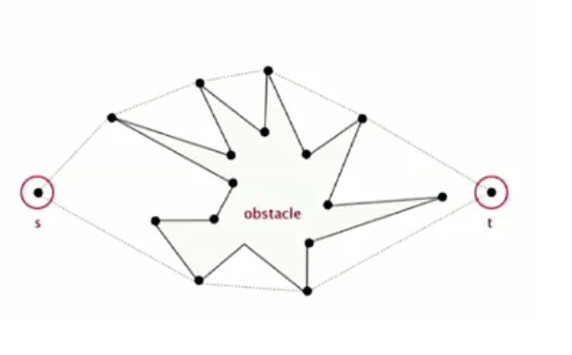
**Introduction:**

Let S be a set of points in the plane.The convex hull of S is the smallest convex polygon that contains all the points of S.



**Applications:**

1. Path finding in robots when there are obstacles along the way.



The shortest path for robot is along one of the set of hull points if obstacle is along the way .

2. Visual Pattern matching

The convex hull of the image is created to make a pattern which is used to identify number or letter. For example license plate.

3. Wide Applications

The problem of finding convex hulls finds its practical applications in [pattern recognition](https://en.wikipedia.org/wiki/Pattern_recognition), [image processing](https://en.wikipedia.org/wiki/Image_processing), [statistics](https://en.wikipedia.org/wiki/Statistics), [geographic information system](https://en.wikipedia.org/wiki/Geographic_information_system), [game theory](https://en.wikipedia.org/wiki/Game_theory), construction of [phase diagrams](https://en.wikipedia.org/wiki/Phase_diagrams), and [static code analysis](https://en.wikipedia.org/wiki/Static_code_analysis) by [abstract interpretation](https://en.wikipedia.org/wiki/Abstract_interpretation)

**Serial algorithm:**

The algorithm used here is the Jarvis march which is as follows:

1. The leftmost point is found out which becomes the first hull point.
2. Select any random candidate-in-run hull point other than the first hull point and traverse through all the points and for each point find orientation previous hull-point->candidate-in-run-hull-point->traverse-point

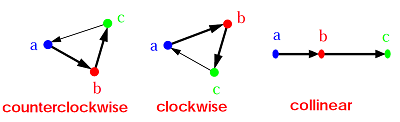
If counter clockwise :

Change the candidate-in-run hull point to traverse->point

3. If the candidate-in-run hull point is not the first hull point set the previous

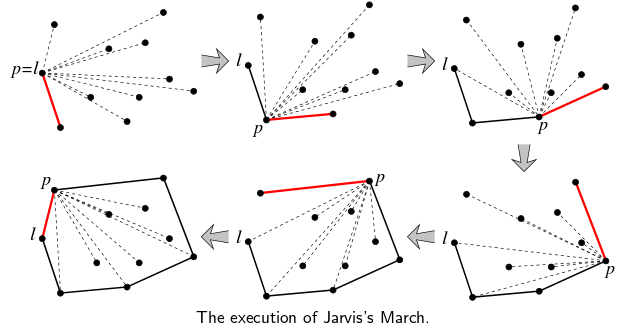
hull point to candidate-in-run-hull point and repeat the step 2 after adding

hull point to set of hull points.



Counter Clockwise -> Left Turn

Clockwise -> Right Turn



**Complexity of algorithm:**

The main task of the algorithm is to find the hull points until all the other points are enclosed within the formed hull.

So the complexity of algorithm is O(n\*h) where n is the number of input points and h is the no of hull points.

The no of hull points can vary between n^⅓ (circle enclosing points)to n^2 (all the points on the hull)

So the worst case complexity is O(n\*n) and average case is O(n^4/3).

**Scope of parallelisation:**

The task of finding the hull points can be divided into sub tasks of finding intermediate hull points which are not dependent on each other. So there is no task dependency. Moreover the result of any task is not used in any other task so there is no data dependency.

**Effect of increasing problem size**:

On increasing the problem size, the task of finding hull points has more points to take into account and also there are obviously more hull points that need to be included so number of computations for finding the orientation of the points to get the hull point at each step and so do the number of steps because of higher number of convex hull points.

**Parallelisation Strategy:**

The task is to find the next hull point by finding the orientation of the given point with respect to the previous hull point and candidate-in-run hull point.

So the task is data oriented and so we use work sharing construct to divide the points among the threads launched on each core. Each core has its set of points and initial hull point (same for each thread initially). Then at each step of finding hull point the suitable hull point for each thread is found out. The hull points obtained locally for each thread are then again checked for orientation with the previous hull point and the candidate-in-run hull point (which is one of these locally obtained hull points) to get the final next hull point.

In next step, this hull point becomes the previous hull point and the same task is repeated.

**PseudoCode:**

In the serial algorithm

1. Initialize next Hull Point as any point for all threads.
2. For all threads check orientation of all other points and previous hull point against that thread’s next Hull point.
3. Find the winner of all the thread points according the orientation
4. Add the winner as the last hull point

**Improving Cache performance:**

* In the given algorithm we can use two arrays to hold x and y coordinates respectively.

**Scope of improvement in performance**

* We can see that x and y value of a particular points are always accessed together. Hence instead of defining two different arrays we can use a 2D array to store x and y coordinates as int point[num\_points][2]
* Hence we will improve the cache performance of our code by reducing the memory access time.

**Results:**

Input:

n=7

{0, 3}

{2, 2}

{1, 1}

{2, 1}

{3, 0}

{0, 0}

{3, 3}

Output:

Number of points in Polygon = 4

(0, 3)

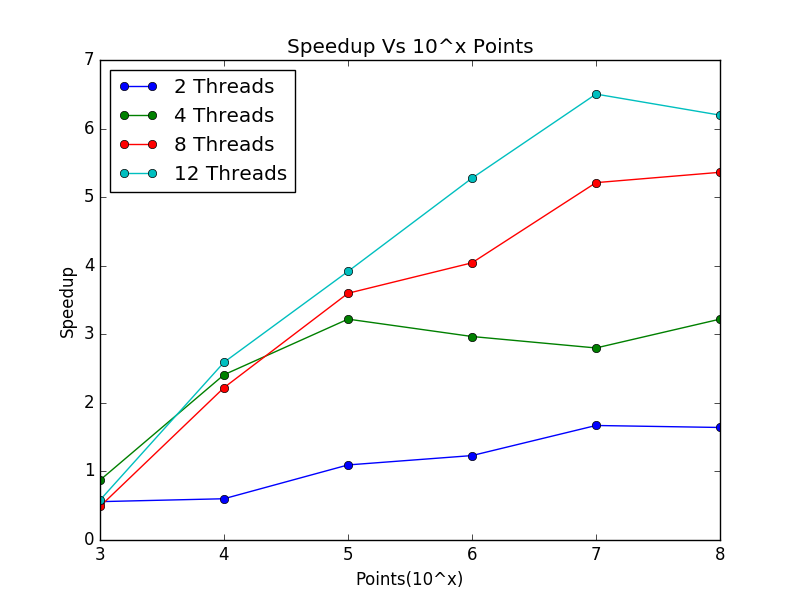
(0, 0)

(3, 0)

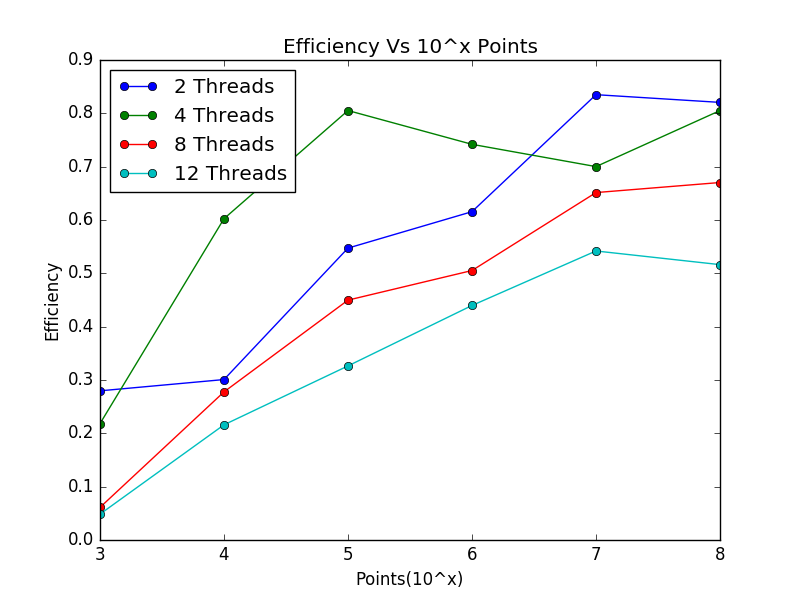
(3, 3)

**Performance :**

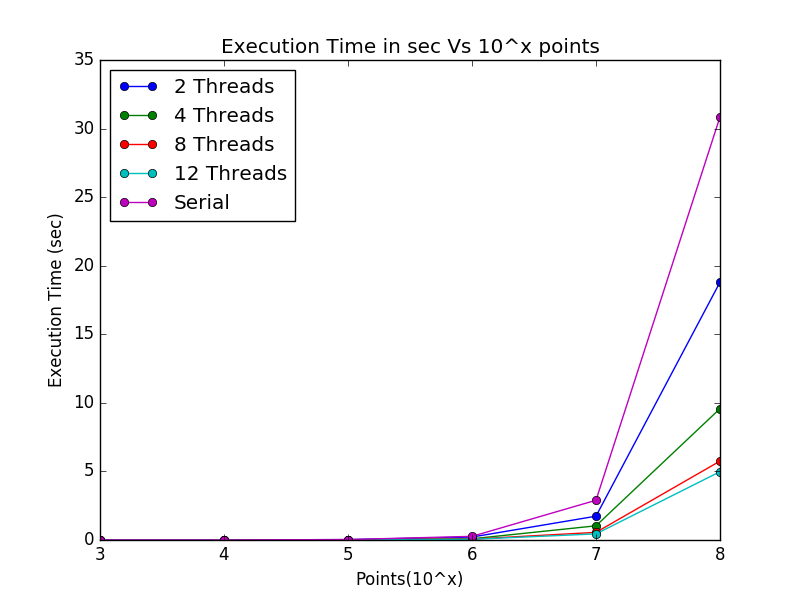
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Problem Size(10^x)** | **Parallel n=2** | **Parallel n=4** | **Parallel**  **n=8** | **Parallel**  **n=12** |
| **3** | 0.5593928790414414 | 0.8727952608786193 | 0.48906512116455986 | 0.5819184111851248, |
| **4** | 0.6016715993670485 | 2.4098186050278736 | 2.2216776094074735 | 2.591400184514559 |
| **5** | 1.0946002062633584 | 3.2210191692635615, | 3.598116508163498 | 3.916965912339677 |
| **6** | 1.231210409190138 | 2.9675793420827326 | 4.042222821445817 | 5.276893185424406 |
| **7** | 1.6699646829664792 | 2.80045809897227 | 5.211414508818344 | 6.50293037195239 |
| **8** | 1.641038015234306 | 3.2207576931942783 | 5.362280352288761 | 6.195655728361214 |



The speed up for small problem-size is lesser than 10^5 owing to the synchronization overheads. Parallel slowdown is happening in this case where the benefit due to parallelization is lesser than the overheads involved.



The Efficiency increases as problem size increases as problem size increases. This is obvious from the Gustafson’s law which says that as problem size increases the speedup increases because the time for which code runs serially decreases which leads to increase in efficiency for fixed no of processors. Gustafson’s law says that the true parallel power of a large multiprocessor system is only achievable when a large parallel problem is applied.



**Karp: Flatt Metric analysis:**

The fraction of serial code experimentally determined for number od point equal to size 10^7:

e=(1/-1/p)/1-1/p

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Value of e | p=2 | p=4 | p=8 | p=12 |
| Problem Size : 10^7 | 0.19763011781018935 | 0.1427792476128514 | 0.0764416945167199 | 0.0768474478375426 |

As we can see, the serial fraction decreases as we go from p=2 to p=8 which indicates that the code is embarrassingly parallel and the serial fraction of code decreases which explains the increase in speedup and efficiency. However for p=12, we see that serial fraction remains almost constant indicating the serial portion has started dominating and the code is no longer parallelizable for more no of cores.

**Observation and Conclusion:**

We observe that as problem size increases the speed up increases. This is because in our algorithm the expected ideal speed up is O(h\*n/p) and as p increases considerably against the number of points we get a good speed up.

However, after a limit, the problem becomes difficult to run on more number of processors and hence speed up stops increasing considerably.

**Future scope** :

As a future scope, we can divide the plane in *p* parts and then combine it using different algorithm.

This will give a more speed up than our current algorithm.

As even further extension, we can change the algorithm to Graham Scan Algorithm, which is more time-optimized algorithm, and then parallelize the code,

**References:**

Applications:

* <http://www.tcs.fudan.edu.cn/rudolf/Courses/Algorithms/Alg_ss_07w/Webprojects/Chen_hull/applications.htm>

Convex hull and algorithm:

* <http://mathworld.wolfram.com/ConvexHull.html>
* <http://www.ti.inf.ethz.ch/ew/courses/CG13/lecture/Chapter%203.pdf>
* http://www.cse.buffalo.edu/faculty/miller/Courses/CSE633/Vertlieb-Fall-2012-CSE633.pdf