IN3030 Oblig 5

1. Intro

In this oblig, we are introduced to Convex-hull algorithm. Our goal was to write a sequential ConvexHull program and then try to parallelize it and achieve speedup >1 for some of $n = 1000, 10000, 1000000\ and\ 10000000$ values. I will in the following text describe how I did this, present my results, and discuss the findings.

2. User guide

To run the program (after compiling all the files), simply run the following command format in the terminal:

```
java ConvexHullPara2 <n> <seed>
```

where n is marking the number of points wanted generated and seed is the staring seed value for random points to be generated from.
Example:

```
java ConvexHullPara2 1000000 10
```

If the n value is < 200, then the program will run in the visual test mode where both parallel and sequential ConvexHull lists will be printed, and a drawing of the result presented in a pop-up window. In this way, one would be able to visually confirm if the result is correct.

3. Parallel version – how you did the parallelization

I followed the instructions given in the oblig text on how one might do the parallelization and did following (Diagram 1):

- Sequentially find MIN_X and MAX_X points.
- Main thread then starts two new threads and gives an empty list to store points to each thread and the section of all points where to search (above the min-max line or below). In this first case this would be all points. Each thread receives a number marking its level down the thread tree. (In this case Main-level 1, Th1-lvl2, Th2-Lvl2). Main thread then waits for both threads to finish and adds MAX_X point, appends right thread list, adds MIN_X point and appends left thread list to overall result list.
- Each thread then figures out if there is a furthest point in the corresponding sector examined by the thread. If yes then, if the maximum number of nodes on the current level does not exceeds the thread count, a new child-thread is started and delegated right sub-section of the points in which to search while the old thread continues working on the left portion of the points. Old thread waits for the child thread to finish and adds points and appends the lists in the same fashion as the Main thread above.
  If, however, the thread count number is exceeded, then the thread finishes all the work by itself using the sequential version of ConvexHull algorithm that was provided to us.

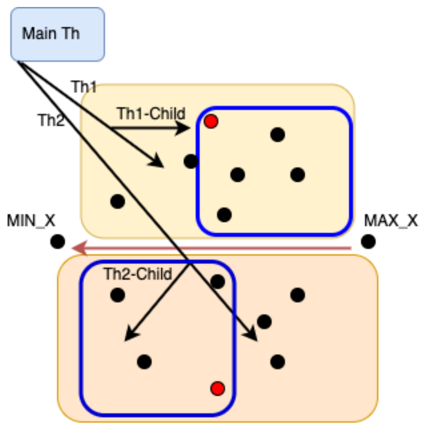
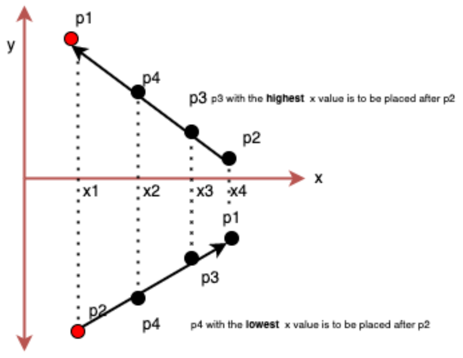


Diagram 1.

4. Implementation

Sequential solution was given to us. I'll just describe how I tried to solve the problem of adding the points on the line in the correct order.
After my recursive solution to this problem was slow (20sec for n=10mil) I reverted to sorting the points on the line using the x coordinate value if the line is horizontal and y value of the point if the line is vertical. However, there is a complication. If we sort based on point 2 x value and the line is pointing from point 2 to point 1 like this (p1<-----------p2), then when the line points in the other direction (p2----------→p1) we get the wrong result. So, one needs to check which point has the highest x value and then sort based on that points x value. Same is true for vertical line. Hopefully the diagram 2 below should make things a bit clearer.


y
p1
p4
p3 p3 with the highest x value is to be placed after p2
p2
x
x1
x2
x3
x4
p1
p3
p2
p4
p4 with the lowest x value is to be placed after p2
Diagram 2

I could, however, be wrong in making all these assumptions and it is possible that my solution doesn't calculate the right order of points. I tried testing it the best way I could, and it looked ok but then again, I might be wrong.

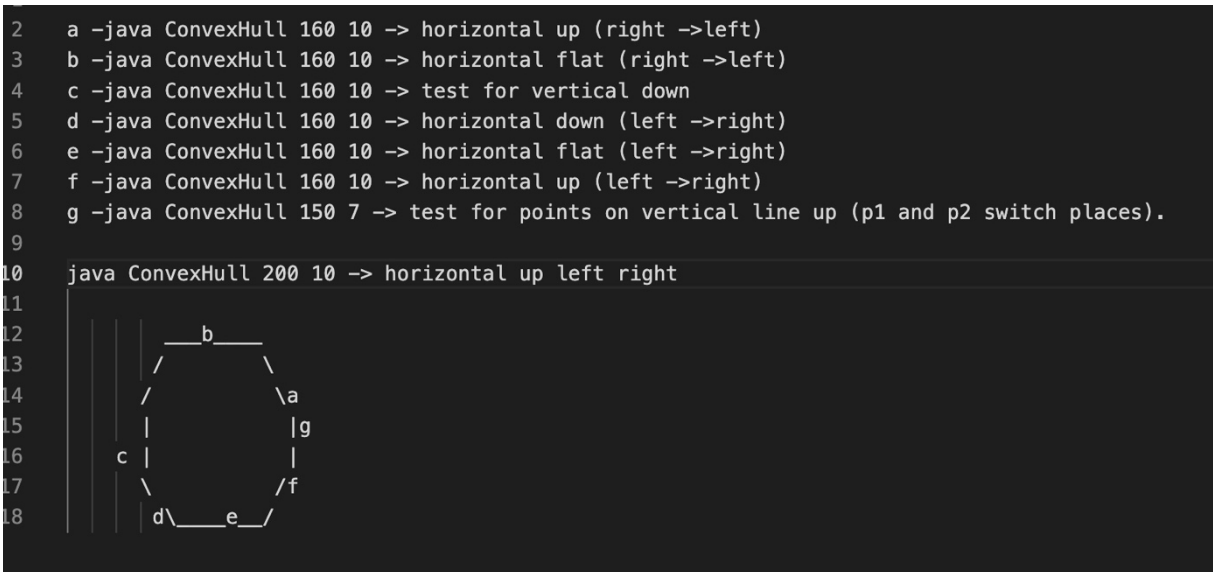
The whole program is run through ConvexHullPara2 main method. This class extends the sequential ConvexHull class. QuickHullPar method of the ConvexHullPara2 starts the initial 2 threads and each run method is responsible for starting the child-threads or continuing as described above.

Testing

Sequential solution was given so I tested how points on the same line were added visually by inspecting different n and seed configurations up to n=200. Later I compared the sequential and parallel convexHull lists and asserted that they are the same.
I've managed to identify a few configurations that have the right number of points on the same line in all directions:
Horizontal-ish right->left up/down

Horizontal-ish left->right up/down
Vertical up->down
Vertical down->up

Here are the configurations:



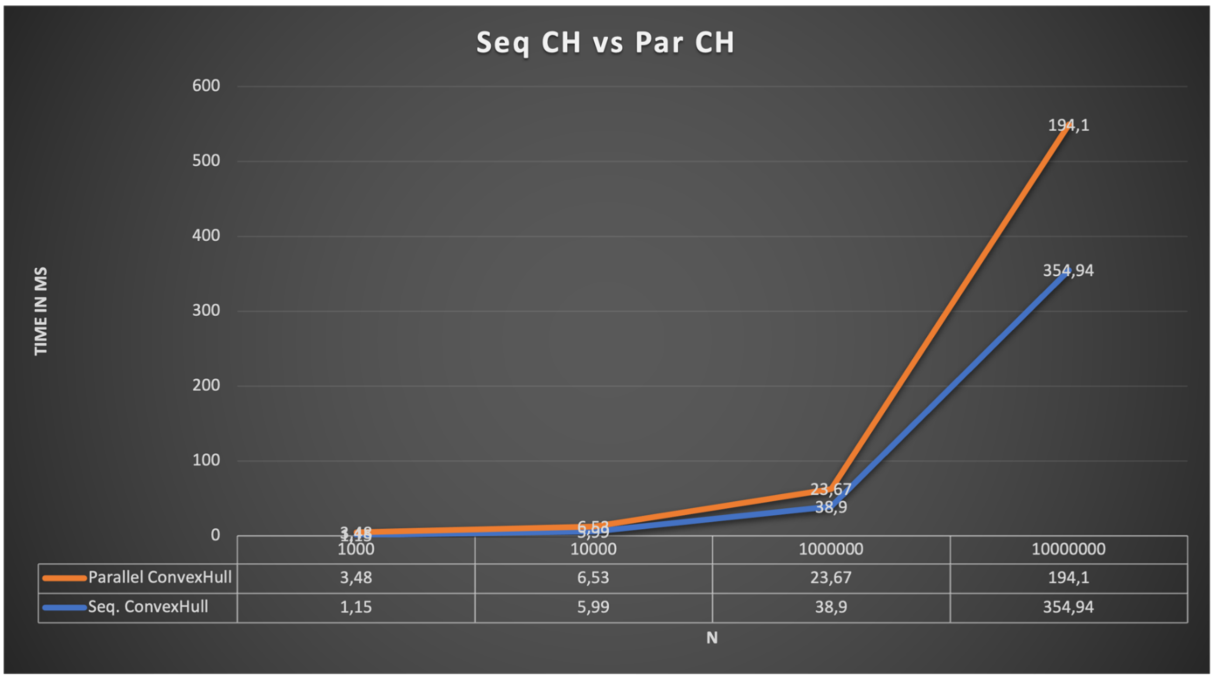
```
a -java ConvexHull 160 10 -> horizontal up (right ->left)
b -java ConvexHull 160 10 -> horizontal flat (right ->left)
c -java ConvexHull 160 10 -> test for vertical down
d -java ConvexHull 160 10 -> horizontal down (left ->right)
e -java ConvexHull 160 10 -> horizontal flat (left ->right)
f -java ConvexHull 160 10 -> horizontal up (left ->right)
g -java ConvexHull 150 7 -> test for points on vertical line up (p1 and p2 switch places).

java ConvexHull 200 10 -> horizontal up left right

          ___b____
         /        \
        /          \a
        |           |g
      c |           |
        \          /f
         d\____e__/
```

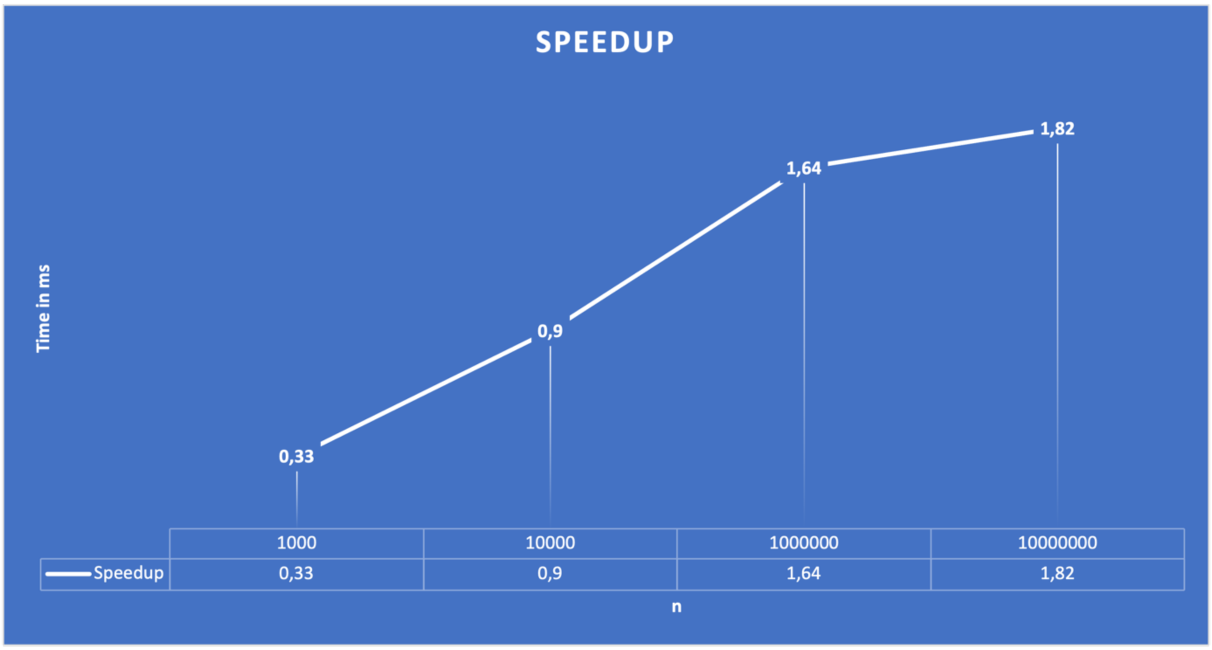
5. Measurements

I ran the program on my MacBook Pro running on 3,1Ghz with 2 cores and 8Gb RAM.



	1000	10000	1000000	10000000
Parallel ConvexHull	3,48	6,53	23,67	194,1
Seq. ConvexHull	1,15	5,99	38,9	354,94

Graph1.



	1000	10000	1000000	10000000
Speedup	0,33	0,9	1,64	1,82

Graph 2. Speedups in ms for n 1000, 10k, 100k, 10m.

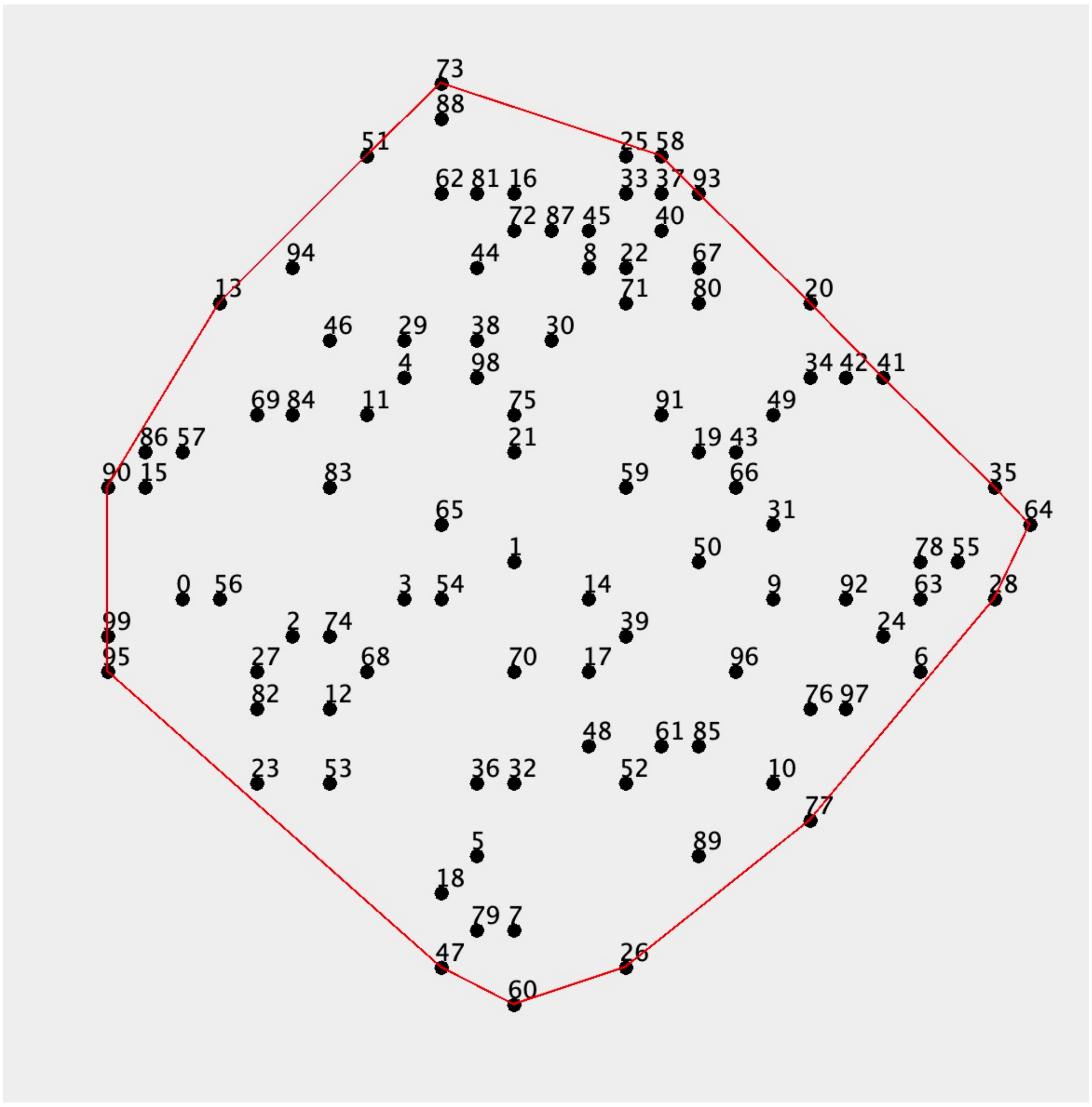
Looking at the results (Graph 1 and Graph 2) we notice that speedups >1 are achieved in cases where n is 1mil and 10 mil. I would like to add here that I only noticed that the parallel version is faster when I placed everything in the loop for 7 iterations! Running things only one time showed that the parallel solution is slower! Moreover, I tried changing the strategies for when the new threads should be initiated. Tried to reduce the number of available threads until 0. Results were pretty much the same as using the level-based strategy, but this might be because I only have 4 logical cores. Level based strategy seems as more balanced, but it might lead to having a bit more threads than the logical cores. The parallel version is thus faster because we divide point pool, and each thread is independent. Chokepoints that cost time are where we wait for child threads to finish.
Worth noting is that in my implementation I do not parallelize sorting the points on the same line.

Conclusion

In this last assignment we explored how algorithms (Convex hull) could be paralyzed through recursion. I've tried some different strategies to mixed results and settled on the level-based thread initiation. Speedups over 1 were achieved for n values 1mil and 10 mil.

Appendix

One hundred points for seed 10.


73
88
51
25 58
62 81 16
33 37 93
72 87 45
40
94
44
8 22
67
13
71
80
20
46 29 38 30
4 98
34 42 41
69 84 11 75 91 49
86 57 21 19 43
90 15 83 59 66 35
65 31 64
1 50 78 55
0 56 3 54 14 9 92 63 28
99 2 74 39 24
95 27 68 70 17 96 6
82 12 76 97
48 61 85
23 53 36 32 52 10
77
5 89
18
79 7
47 26
60