# Oblig 5 – IN 3030

Marcusti@student.matnat.uio.no

## Introduction

In this Oblig we were to find the convex envelope of a given group of points in a 2D-space. We were to implement a sequential as well as a parallel version, and archive a speedup greater than 1 for large N.

## User Guide

To compile the program, run the command 'javac *.java'. To run the program, run the command "java Main <N> <K> . So if you want to find the convex envelope of 1 million generated points, running on all available cores you would write –

"Java Main 1000000 0".

## Parallel Convex Envelope

The file Oblig5.java contains both the sequential and the parallel implementation. In a short description, the algorithm works by drawing a line between two points, and a adding the point which is furthest away from the given line. Using some recursive magic, you end up with a list of all points which are located at the outer bounds of all points generated – a convex envelope.

I've chosen to implement the parallel algorithm by method 1), which works by first dividing all points between K-amount of Threads. Now these threads will use the sequential solution and find each their own convex envelope. Once all threads are done, we merge these separate convex envelopes into a large list. This list will now contain much less than the initial N points, as all points in the combined list already are potential points for some outer region. Finally I make the main-thread run the sequential solution on this list of candidate-points. Since there are way less than N points, the sequential calculation will be relatively quick. In the end we end up with a convex envelope, containing the correct solution as running the sequential solution on all N points.

## Implementation

I began with implementing the sequential solution, inspired by the class slides and group sessions. I did my testing with the TegnUt class, which prints a visual result of the algorithm. I began with implementing the top-half of the convex region. Once I got it working, it was easy to repeat my work on the bottom half of the convex region.

I initially ran into some RAM problems where my computer was running out of memory every time I ran for large values of N. The problem was how I was decreasing the amount of points to search through for each recursive call, and how many times we should search recursively. This was fixed by implementing more helper – methods, in addition to only keeping points which are on the outside of the current "line".

The parallel solution is not the optimal way of solving this problem; however it proves useful as it achieves speedup of up to 1.3.

# Measurements.

I've run the algorithm on my laptop consisting of a:

i7-7500U CPU @ 2.70GHz – Dual core/2 threads per core for a total of 4 Threads. My memory includes 8GB of ram, running at 1867MHz. In addidtion to L1 cache of 128kB, L2 cache 512kB and L3 cache 4MB.

N = 100

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java Main 100 0
Sequential Algorithm Running...
      0.167577ms
      0.058115ms
      0.064797ms
      0.076587ms
      0.078712ms
      0.071589ms
      0.069732ms
--------------------------------------
      Median Time Sequential
           0.071589ms
--------------------------------------
Parallel Algorithm Running...
      1.436581ms
      0.414913ms
      0.288831ms
      0.234446ms
      0.477135ms
      0.27407ms
      0.279035ms
--------------------------------------
      Median Time Parallel
           0.288831ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          0.2479
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
Omkrets:

29, 42, 15, 78, 17, 99, 49, 53, 88, 6, 19, 95, 65,
```

N = 1000

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java Main 1000 0
Sequential Algorithm Running...
      0.735003ms
      0.526386ms
      0.321505ms
      0.177186ms
      0.159507ms
      0.185348ms
      0.175732ms
--------------------------------------
      Median Time Sequential
           0.185348ms
--------------------------------------
Parallel Algorithm Running...
      1.357549ms
      0.470833ms
      0.516176ms
      0.508084ms
      0.511416ms
      6.511427ms
      0.525312ms
--------------------------------------
      Median Time Parallel
           0.516176ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          0.3591
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
Omkrets:

485, 290, 396, 896, 782, 178, 172, 433, 466, 228, 341, 836, 588, 789, 212, 83, 275
, 551, 826, 840, 712, 311, 326, 62, 432, 196, 214, 98, 154, 615, 643, 922, 828, 39
4, 181, 783, 67, 721, 938, 932, 69, 682, 237, 545, 732, 569, 110, 785,
```

N = 10 000

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java Main 10000 0
Sequential Algorithm Running...
      4.210726ms
      1.488883ms
      1.504403ms
      1.935568ms
      1.879242ms
      1.750643ms
      2.239301ms
--------------------------------------
      Median Time Sequential
           1.879242ms
--------------------------------------
Parallel Algorithm Running...
      5.128792ms
      5.327479ms
      1.753059ms
      1.437727ms
      1.374346ms
      3.097439ms
      3.128571ms
--------------------------------------
      Median Time Parallel
           3.097439ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          0.6067
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
```

N = 100 000

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java Main 100000 0
Sequential Algorithm Running...
      26.221493ms
      11.135455ms
      7.383601ms
      8.248133ms
      8.471438ms
      8.274658ms
      7.589959ms
--------------------------------------
      Median Time Sequential
           8.274658ms
--------------------------------------
Parallel Algorithm Running...
      12.456726ms
      8.839654ms
      9.010994ms
      17.335995ms
      9.946617ms
      7.18142ms
      7.656126ms
--------------------------------------
      Median Time Parallel
           9.010994ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          0.9183
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
```

N = 1 Million

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java Main 1000000 0
Sequential Algorithm Running...
      93.527118ms
      60.956788ms
      50.632184ms
      41.240622ms
      55.079406ms
      70.48452ms
      48.648965ms
--------------------------------------
      Median Time Sequential
           55.079406ms
--------------------------------------
Parallel Algorithm Running...
      96.415034ms
      68.339044ms
      41.559273ms
      43.919301ms
      34.572484ms
      35.162193ms
      35.849694ms
--------------------------------------
      Median Time Parallel
           41.559273ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          1.3253
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
```

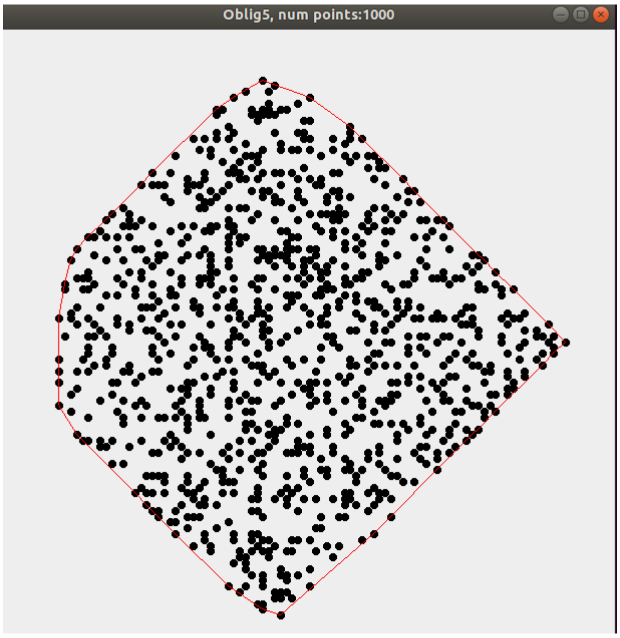
N = 10 Million

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java Main 10000000 0
Sequential Algorithm Running...
      563.817346ms
      533.479292ms
      508.476683ms
      509.933994ms
      490.706679ms
      519.143256ms
      487.362611ms
--------------------------------------
      Median Time Sequential
           509.933994ms
--------------------------------------
Parallel Algorithm Running...
      406.044631ms
      399.896497ms
      382.491937ms
      387.555054ms
      365.985302ms
      398.74261ms
      389.297797ms
--------------------------------------
      Median Time Parallel
           389.297797ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          1.3099
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
```

N = 100 million.

```
mvrcus@Mvrcus:~/Documents/IN3030/Oblig5$ java -Xmx7300m Main 100000000 0
Sequential Algorithm Running...
      6259.525996ms
      6253.178313ms
      6209.81436ms
      6238.875007ms
      6421.284507ms
      6325.933349ms
      6084.943855ms
--------------------------------------
      Median Time Sequential
           6253.178313ms
--------------------------------------
Parallel Algorithm Running...
      4853.917119ms
      4752.337699ms
      4840.778946ms
      4656.272185ms
      4776.358502ms
      4892.318064ms
      5344.375378ms
--------------------------------------
      Median Time Parallel
           4840.778946ms
--------------------------------------
--------------------------------------
      Speed Up S/P
          1.2918
--------------------------------------
--------------------------------------
 The Sequential and Parallel solutions
 generated the same results.
--------------------------------------
```

TegnUt av N = 1000




Convex Envelope consists of these points:

485, 290, 396, 896, 782, 178, 172, 433, 466, 228, 341, 836, 588, 789, 212, 83, 275, 551, 826, 840, 712, 311, 326, 62, 432, 196, 214, 98, 154, 615, 643, 922, 828, 394, 181, 783, 67, 721, 938, 932, 69, 682, 237, 545, 732, 569, 110, 785.

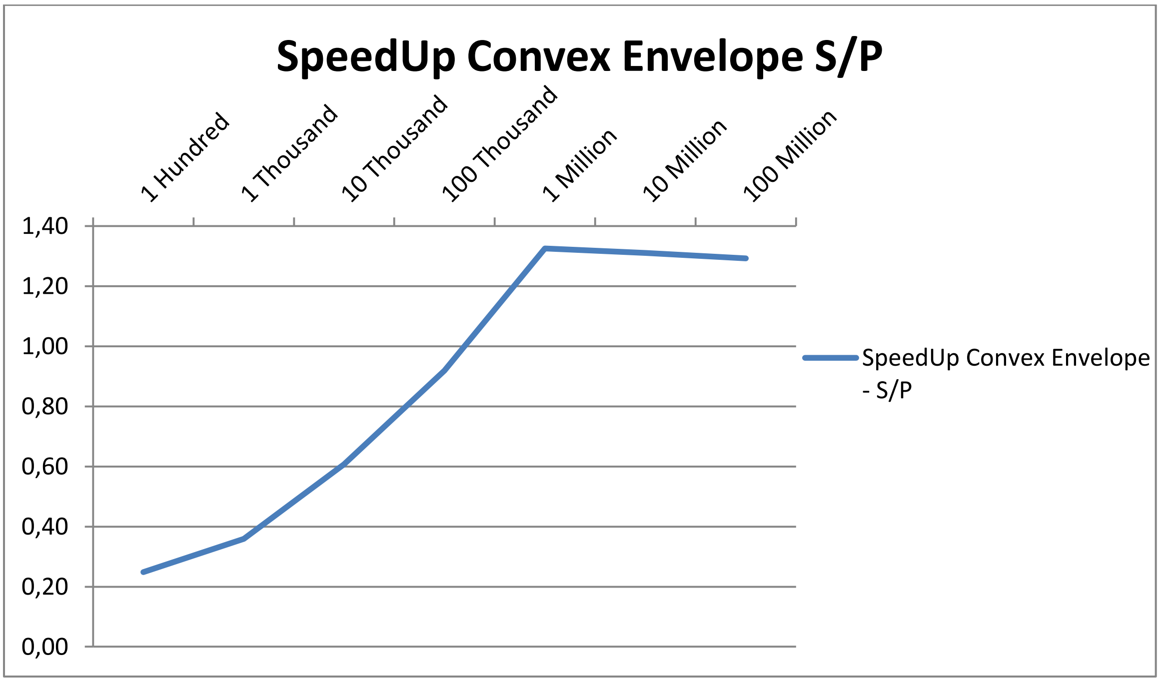



**Figure 1: Y axis shows the speedup, S/P for the convex envelope sort.**

## Conclusion

As we can see from the graph, it is clear that the parallel implementation is the superior solution when N is larger than around 100 000.  We can also see how the speedup seems to stop increasing at around 1.3, this is because of Amdahls Law. There will always be some work that has to be serialized. This in addition to the overhead of communication and creation of threads proves that even with 4 cores we cannot realize a speedup of 4.0.

## Appendix

The output of my program is the result of running each algorithm 7 times and  comparing the median of these runs. I've also included a check to see if the parallel version creates an equal solution to that of the serial implementation.