# IN4330 Oblig 4

Computing specifications used for initial runnings:

- Processor is an Intel(R) Core(TM) i5-9400F , 2,90 GHZ with 6 cores and 6 logical processors

## Usage

The program to be runned is the Oblig4.java which runs convex Hull in sequentially and in parallel, runs it 7 times, collects tests results, and finally draws the outcome.

General use:  java Oblig3.java <n>  <seed> <number of threads t>

Usage example

```
IN4330\Oblig4 on  main [$!?] via  v17.0.10 took 28s
❯ java oblig4.java 10000 23 6
```

Run example

```
IN4330\Oblig4 on  main [$?] via  v17.0.10 took 9s
❯ java oblig4.java 10000000 23 12
-------------------------[RESULTS]---------------------------------
Median time of 7 runs for sequential Convex Hull: 748.873599 ms
Median time of 7 runs for Parallel Convex Hull: 146.739999 ms
Speed up of convex hull for 7 runs: 5,103
Test run for n=10000000, and number of threads t=12

-------------------------------------------------------------------
```

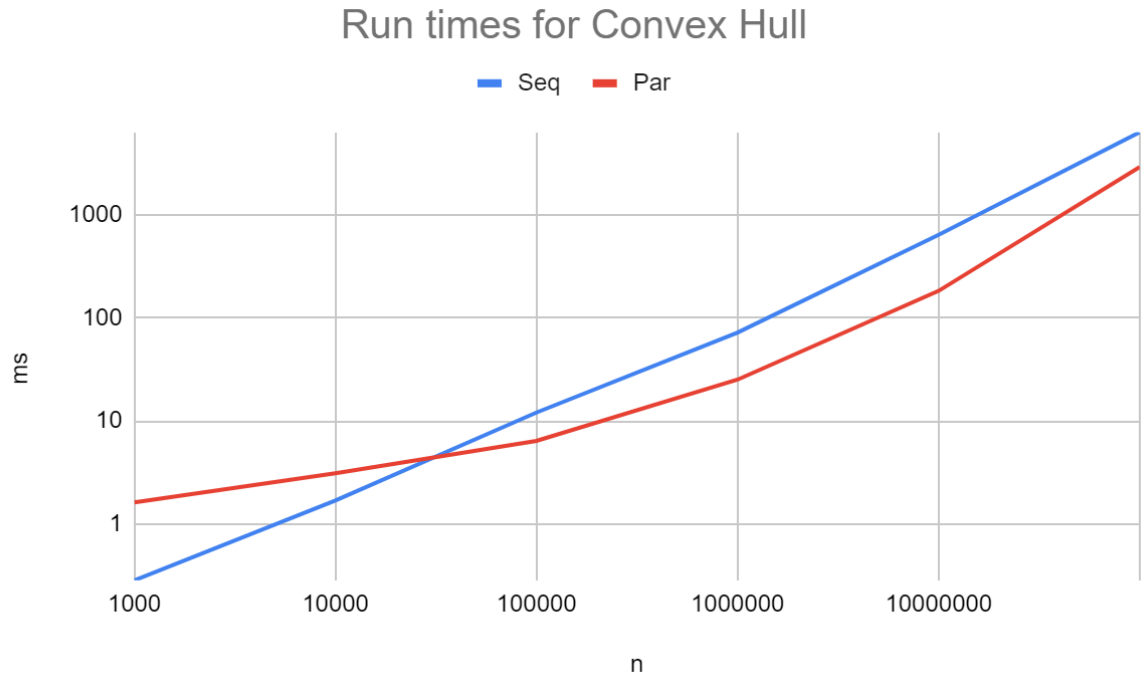
## Parallel solution

I ended up implementing a variation of the second method for solving convex hull in parallel. I achieved this by dividing the initial points arbitrarily  into equal amounts across threads. Then each thread does the sequential algorithm and attempts to find a set of utmost points for each thread. Each thread fills a local kohyll list with these points. When all threads are done finding points, we have a set of lists with points that most of the time either at the convex or much further from the center then our initial point. From here we make append the list points each thread found to a new list.  From here,  we run the sequential algorithm one last time using the new list with the appended points , but now we have substantially reduced the number of points to look through. Furthermore, most of the points we are now working with are most of the time closer to the convex points than the initial set of points, which makes the recursion end much faster.

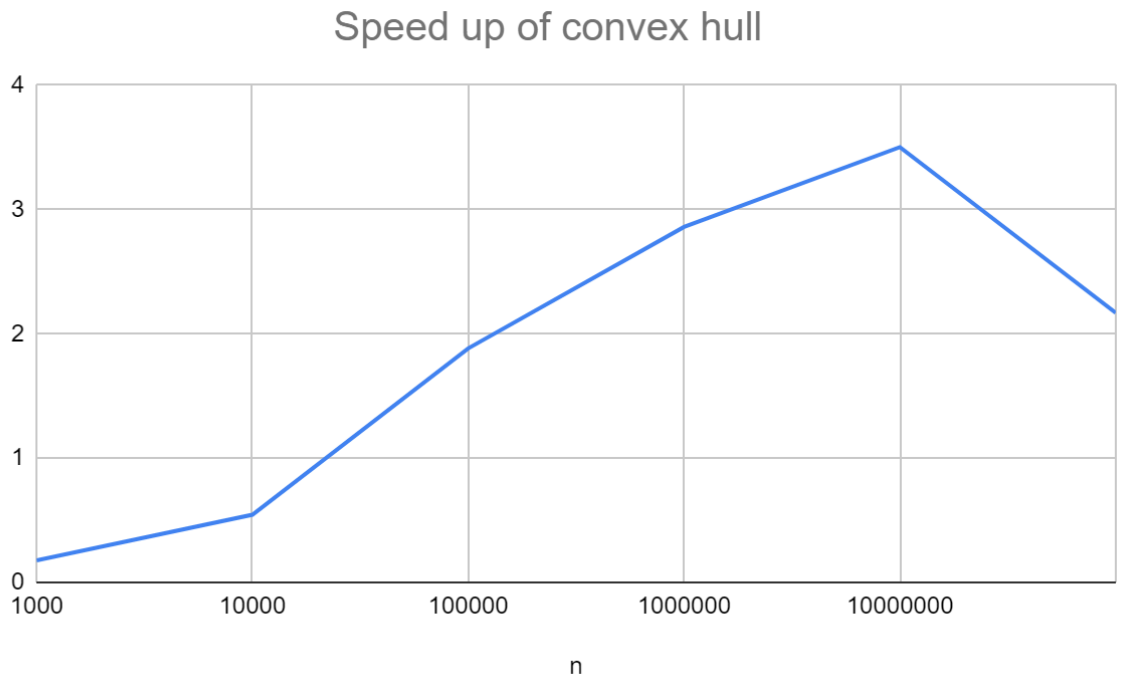
# Test results

The following results were done with the specifications mentioned, using the max available threads which is 6.

| Run times convex hull in milliseconds, using 6 threads | | |
|---|---|---|
| N | Seq | Par |
| 1000 | 0.2844 | 1.6202 |
| 10000 | 1.6939 | 3.1129 |
| 100000 | 12,074 | 6.4139 |
| 1000000 | 71.9999 | 25.172 |
| 10000000 | 639.6313 | 182.6646 |
| 100000000 | 6318.4777 | 2911.4738 |

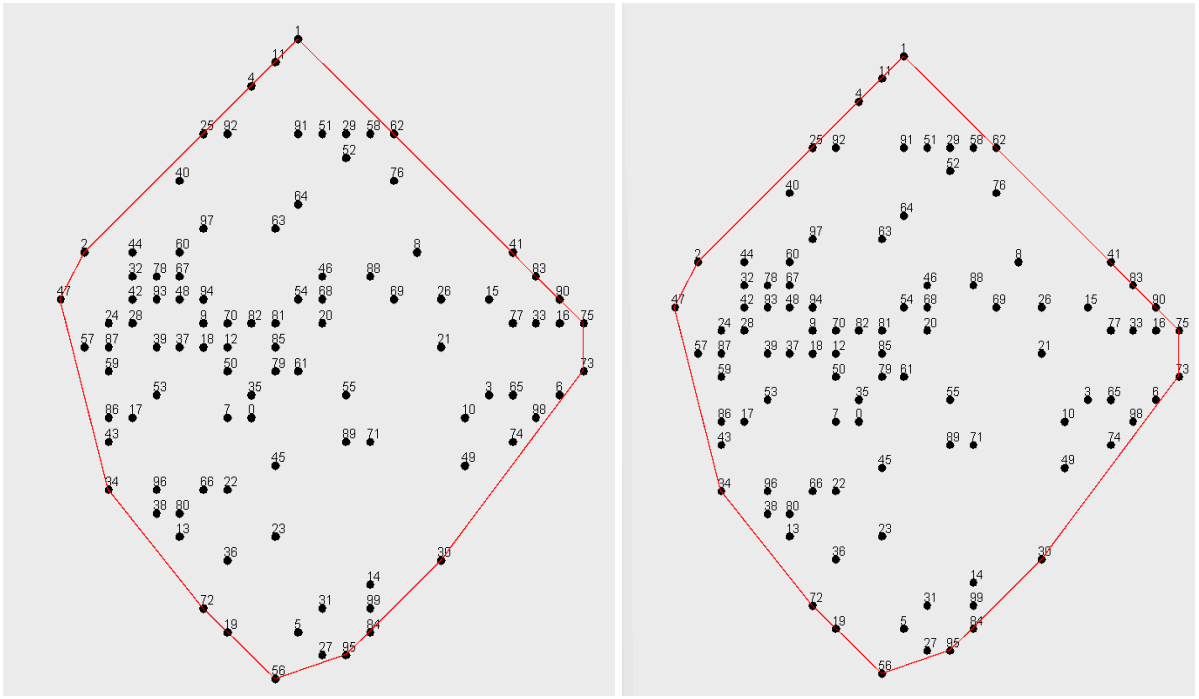


| Speed up of convex hull | |
|---|---|
| N | |
| 1000 | 0,176 |
| 10000 | 0,544 |
| 100000 | 1,883 |
| 1000000 | 2,860 |
| 10000000 | 3,502 |
| 100000000 | 2,170 |




From the test results we can clearly see that at higher values of n the parallel solution is consistently better than sequential, with speed up increasing all the way up to a hundred million where speedup generally stabilizes around 2-3 speedup.

# Convex hull drawing



Drawing off convex hull of 100 points ran on 6 threads(left seq, right par).

# Discussion and Experiments

## Discussion

One alternative for the second method is to spend a little amount of time dividing the points so that they are at least somewhat close to one another. Currently they are divided arbitrarily, but by having them divided into parts close to one another we can create better consistency in the how close to the convex the points that each thread finds is to the convex. As it currently stands, which you'll see in some of the experiments, the choice in the seed ,number of threads and runtime in general varies heavily between runs which is most likely caused by the way the points are divided up between threads.

Although I didn't choose the first method the oblig text describes, I believe it has the potential to achieve very high speed ups compared to the second method. The tree method works by having several threads created until a certain depth is reached. It's in what decides the stopping of thread generation that I believe there's potential to make a parallel solution that works exceptionally well. One can have a thread cut off, when too many threads are created the recursion then starts. The other one is to have a size cutoff, when the pool of points is divided up to a certain amount, then begin the recursion. With this solution there's a possibility to reuse threads that are done doing work, this where it differs from my solution using a variation of the second method. We inherently have to wait for each thread to finish,

before running the final sequential run. If any threads lags behind then a single thread can be the cause of poor performance times.

## Experiments

Specifications used for experiments:
Processor is an Intel(R) Core(TM) i5-9400F , 2,90 GHZ with 6 cores and 6 logical processors, 16 gb ram

| Speed up of convex hull n= 10000 | |
|---|---|
| Threads | |
| 4 | 1,742 |
| 6 | 1,107 |
| 8 | 1,107 |
| 10 | 1,007 |
| 12 | 1,007 |

| Speed up of convex hull n= 1000000 | |
|---|---|
| Threads | |
| 4 | 2,823 |
| 6 | 3,187 |
| 8 | 3,354 |
| 10 | 3,491 |
| 12 | 4,808 |

I decided to experiment with the number of threads, and how they affect speed ups depending on how many points we are solving the convex hull for.

After experiments with some threads we can come to certain conclusions. On lower numbers of n divided the work between as many threads as possible causes speed up to reduce substantially. This is most likely due to overhead caused by creating many threads, and the potential situation where all other threads are waiting for one slow thread to finish. But on higher numbers of n dividing the large work to as many threads as available reduces the number of points to look through for the last sequential run to the point of speed being increasingly better by using as many threads as possible.