IN3030 Oblig4 report - hermagst@uio.no

User guide

To run the program use the main function inside the `ConvexHull.java` file. The `main` method takes 2 command line arguments: first `n` the number of points to generate, secondly `seed` the seed to use for generating the points using the `NPunkter` class in the precode. Both commandline arguments are expected to be integers. This will benchmark both the sequential and the parallel solution `N_TEST_RUNS` times and report the median run times over the runs. The parallel solution is hardcoded to use every available core - though you could easily change this by changing the initiliazation of the `threads` array in the `ConvexHullPara` constructor to some other value.

Example usage:

```
$ javac *.java

$ java ConvexHull 1000 2024
```

Sequential solution

The sequential solution is a relatively simple recursive solution. It first finds the `argMax` and `argMin` of the x coordinates as we know these will be on the final hull. It then splits into two parts - one for finding the points above the line between the two first points (`visitAbove`) and one for finding the points below it (`visitBelow`). These parts will recursively find all points above and below the initial line.

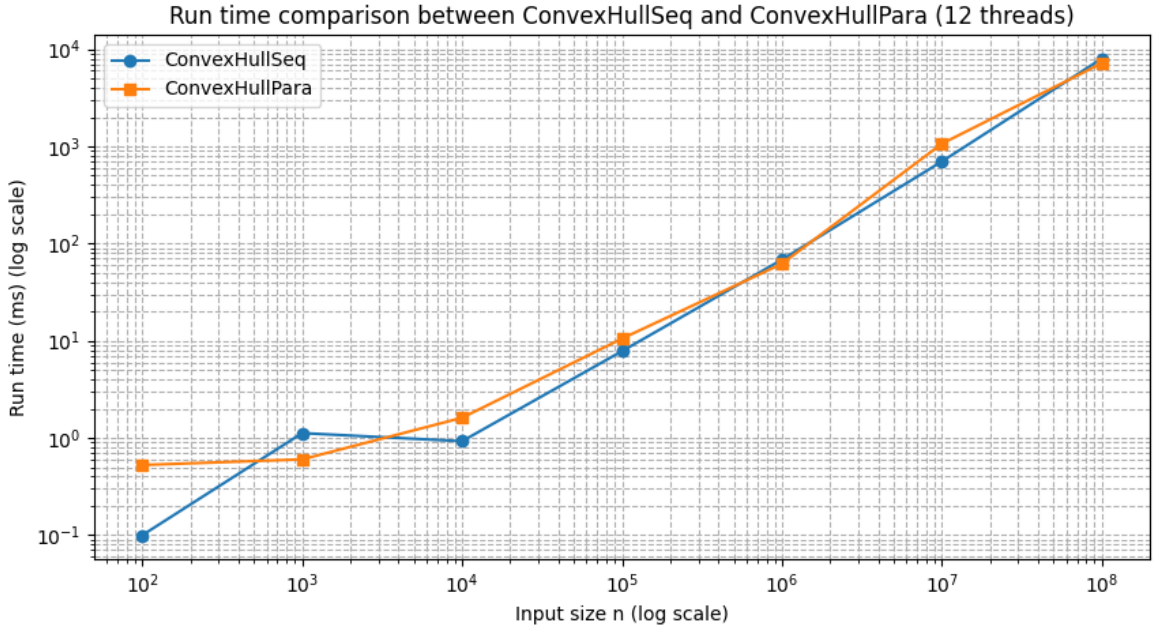
Parallel solution

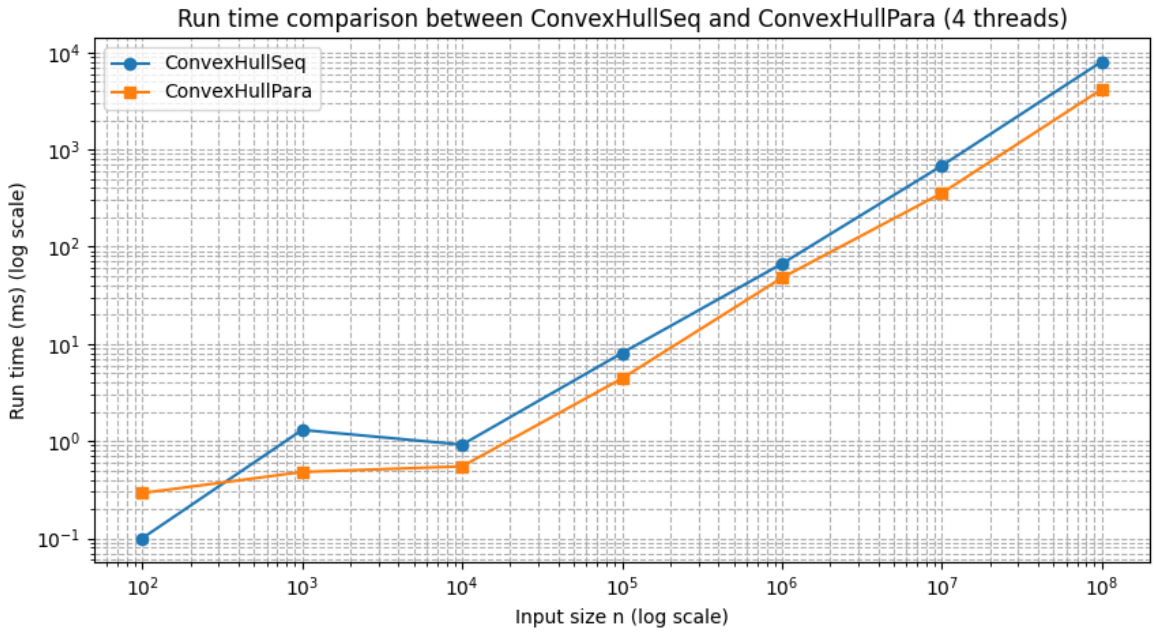
The parallel solution tries to first do a sort of shallow BFS until it has reached a depth where it can start a thread for each branch within the search. It then starts a thread for each branch which does the same recursive search as the sequential solution, only difference being all visited points need to be added to a local visited `IntList` instead of the shared one as this would not be thread safe. When every thread is finished, the main thread then finally sequentially appends all the local visited lists into the shared one and then returns it.

I noticed that the performance of the parallel solution can be quite dependant of the shape of the graph its being run on. For example if there are some sections on the graph that have a lot of points on the hull and some points that have very few, the program will be bottlenecked by the threads working on the areas with many points.

Another weird thing I noticed is that the performance is oddly better when there are fewer threads. For example when initialized with 4 threads I the runtimes were significantly better than when I used all threads on my machine (12). This may be a similar issue to the one described above, where the parallel solution is dependant upon the shape of the graph and if there are many threads we will be doing a lot of unnecessary thread overhead for threads that are doing very little calculating.

Run times


Run time comparison between ConvexHullSeq and ConvexHullPara (12 threads)
ConvexHullSeq
ConvexHullPara
Run time (ms) (log scale)
Input size n (log scale)


Run time comparison between ConvexHullSeq and ConvexHullPara (4 threads)
ConvexHullSeq
ConvexHullPara
Run time (ms) (log scale)
Input size n (log scale)