5th MANDATORY ASSIGNMENT

IN3030 - Efficient parallel programming

NAME: POORIA MEHREGAN
USERNAME: pooriam

---

1. Introduction – what this report is about

This report is about how to achieve an efficient parallel version of the sequential algorithm *Convex Hull*. What you will see is the parallel version is often 3 to 4 times faster for large number of points.

The sequential solution was partially done by the course instructors, what remained to be done in the sequential part, was how to include the remaining points, which are on the straight line, to the convex hull in correct order.

There are results from last run in *materials* folder and a picture to give you an idea and overview of what this algorithm does.

Each algorithm is timed 7 times, and the time presented in *timings.txt* is the median time of these 7 runs.

2. User guide – how to run the program

First make sure you are inside the *src* folder. Then you have 3 options:

I. A full automated testing of these n values : {100, 1000, 10 000, 100 000, 1 000 000, 100 000 000}

```
javac TheMain.java && java TheMain
```

II. n is the number of points, and seed is used to generate a random array of points.

```
javac TheMain.java && java TheMain <n> <seed>
```

III. k is the number of threads

```
javac TheMain.java && java TheMain <n> <seed> <k>
```

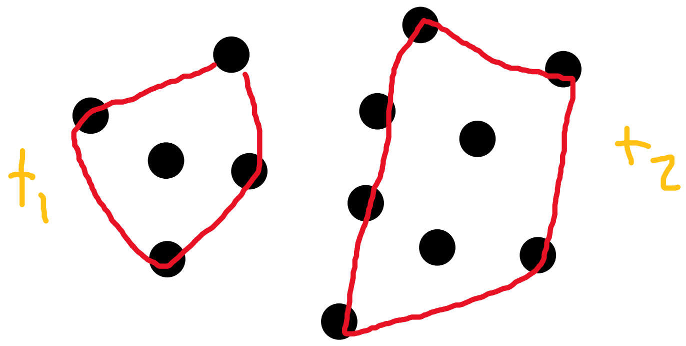
When running option 1, timings and speedup results are saved to a file called timings.txt, which is inside materials folder.

Option 2 and 3 will draw a graph on the screen if n <= 1000 and save the points in the materials folder for n <= 10000.

3. Parallel version – how was the parallelization done

I divided the work between all the available cores on the machine and gave each of them a portion of the points to work with. Each thread will then run the sequential algorithm on the given points and return the resulted convex hull. Thus, each thread eliminates a lot of points by creating its own (little) convex hull.

When all threads are done, the main thread goes ahead and runs the sequential version of the resulted points, and creates the big hull.



In the drawing above you see 2 threads in action, where each thread creates its own little hull. As the result thread 1 eliminates 1 point, and threads 2 eliminates 2 points before returning.

4. **Implementation – how the Java program works & how tested**
The start point of the program is the class *TheMain.java*. This class handles 3 cases, when user gives no input and when user does. If user gives no input, then an automated test of several n values will be executed, and the timing result will be saved to *timings.txt*.
The other two options are explained in section 2 – user guide.

The automatic testing runs each algorithm 7 times for each N value and then times it. When timings are done it finds the median time of these 7 runs and return it, for both sequential and parallel version. This is done because JIT compilation makes the program run faster after several.

The other two options, which are user specified, will run the program only 1 time for each algorithm, times them and print the times and speedups to the console. In addition, it will draw the graph on the screen if n <= 1000 and saves the points in the materials folder for n <= 10000.

5. **Measurements – tables, graphs of speedups, number of cores used**
Machine specification:

Model Identifier:	MacBookPro16,1
Processor Name:	6-Core Intel Core i7 (12 with Hyper-Threading)
Processor Speed:	2,6 GHz
Number of Processors:	1

Total Number of Cores:	6
L2 Cache (per Core):	256 KB
L3 Cache :	12 MB
Hyper-Threading Technology:	Enabled
Memory:	16 GB

To see graphs, tables and curves, please check the materials folder.

6. **Conclusion – just a short summary of what you have achieved**
   The conclusion is that we achieve a better performance where there is needed if we parallelize the sequential algorithms. In this case the parallel version was about 3 to 4 times faster than the sequential one.